UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA
INGENIERIA TÉCNICA EN INFORMÁTICA DE GESTIÓN

DESARROLLO DE UNA APLICACIÓN WEB CON ANGULAR 2 + SPRING + HIBERNATE DE UNA APLICACIÓN DE CONTABILIDAD

Autor: David Blanco París
Tutor: Adolfo Yela Ruiz

Curso: 2015/2016

ÍNDICE

I Introducción y objetivos

Resumen

El objetivo de este proyecto, es el desarrollo de una aplicación web utilizando la tecnología (Spring + hibernate (jpa) + angular 2), aquí juntaremos unos de los framework más populares para trabajar con java + un nuevo framework que está desarrollando google para realizar aplicaciones ricas en el navegador. Lo primero que vamos a realizar es un estudio teórico de los distintos aspectos que conforman esta tecnología y como se integran entre s, para después realizar una implementación de una aplicación web mediante un ejemplo práctico.

En la primera parte vamos a ver una metodología de trabajo de los framework que vamos a utilizar en este proyecto, así como los diferentes aspectos que componentes de este framework..

En la segunda parte, para mostrar la funcionalidad, vamos a desarrollar una pequeña aplicación para llevar por un lado los asientos de una empresa y un pequeño directorio de personas físicas.

Summary

The objective of this project is the development of a web application using technology (Spring + Hibernate (JPA) + angular 2), here will gather some of the most popular framework to work with java + a new framework being developed by google for rich applications in the browser. The first thing we will do is a theoretical study of the various aspects that make this technology as integrated between s, to de-after performing an implementation of a web application using a practical example.

In the first part we will see a working methodology of the framework that we will use in this project and the various aspects of this frame-work that components ..

In the second part, to show the functionality, we will develop a small apply-ing to take seats on one side of a company and a small directory of individuals.

Objetivos

Este proyecto se divide en dos partes, un primer estudio teórico de la tecnología de (angular 2 + spring + hibernate).

Los objetivos de la primera parte se pueden resumir:

- Ver como montar la arquitectura del proyecto.
- Ver como montar la parte de front-end.
    - Arquitectura que monta angular 2.
    - Organización del proyecto.
    - Componentes generales.
- Programación reactiva
    - Librería rxjs
    - Paradigma de la programación reactiva
- Arquitectura en la capa servidora
    - Controladores rest.
    - Capa de negocio.
    - Configuración de la transacionalidad.
    - Configuración del módulo de seguridad.
    - Configuración de hibernate (ORM).

Los objetivos de la segunda parte serían la implementación de unos pequeños ejemplos para ver el funcionamiento del modelo teórico.

- Creación de un directorio de personas físicas.
- Mantenimiento de las cuentas contables.
- Mantenimiento del asiento contable.

II Estudio teórico

Arquitectura del proyecto

Las aplicaciones empresariales, actualmente se dividen en dos partes muy diferenciadas, que se pueden desarrollar por separado, podríamos decir que ahora tendremos una aplicación en el cliente y otra distinta en el servidor:

Cliente

En el navegador vamos a desarrollar una aplicación SPA, utilizando las siguientes tecnologías:

- **typeScript**: Tiene implementada parte de javascript 6, además de un sistema de tipos estáticos. Por otro lado, angular 2, recomiendan que se utilice typescript.
- **javascript 6**: Utilizaremos la nueva implementación de clases, el sistema de módulos nuevos definidos.
- **systemjs**: Es la librería que vamos a utilizar para cargar los módulos de manera dinámica.
- **angular 2**: Es el framework de trabajo que vamos a utilizar para desarrollar nuestra aplicación spi.
- **rxjs**: Es la librería que utiliza angular 2, para crear los objetos observables.

Para poder implementar de una manera eficiente esta parte vamos a utilizar las siguientes herramientas:

- **nodejs**: Es el motor v8 de javascript.
- **atom**: Es el editor que vamos a utilizar + el pluging de typescript.

Servidor:

El servidor lo vamos a implementar como una aplicación clásica en 3 capas, aunque la capa de vista será un sistema de servicios rest. Vamos a utilizar las siguientes tecnologías.

- **Spring**: Será el framework que va implementar el contenedor de beans.
- **Spring-security**: Sera el módulo de spring encargado de gestionar la seguridad del proyecto.
- **Spring-mvc**:Es el módulo con el que vamos a implementar los servicios rest y cargar los contenidos tanto estáticos, como dinámicos.
- **Hibernate**: Es un ORM que vamos a utilizar para la capa de persistencia.
- **Java 8**: Es el lenguaje que vamos a utilizar.
- **Tomcat 8:** Será nuestro contenedor de servlet.

Para poder trabajar de una manera eficiente esta parte vamos a utilizar las siguientes herramientas:

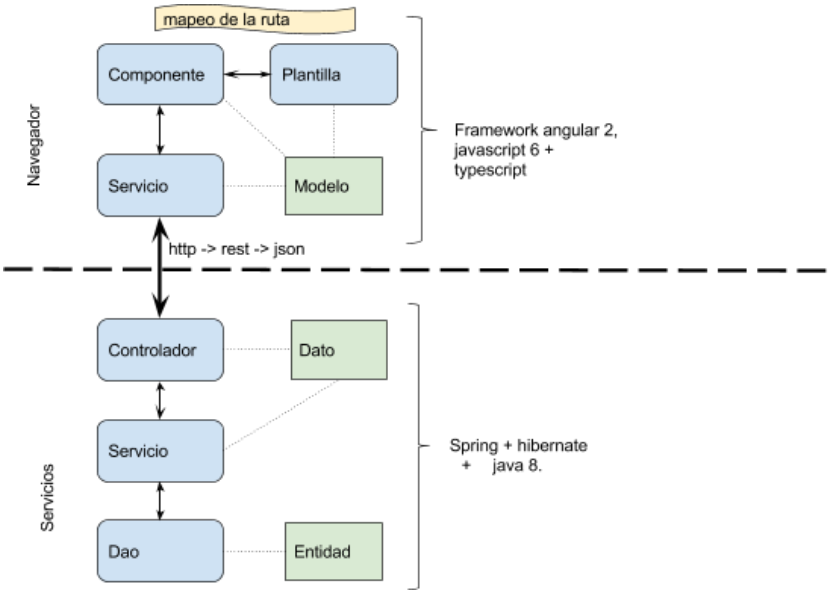
- **eclipse**: como ide para poder trabajar.
- **gradle**: como herramienta para construir el proyecto.

Herramientas comunes

Todo el proyecto estará en el control de versiones github.

Funcionalidad vertical

Nosotros vamos a realizar el desarrollo del proyecto realizando verticales, esto quiere decir que, por cada funcionalidad de negocio, vamos hacer todas las capas de un solo golpe. En el siguiente diagrama podemos ver un corte del vertical.



En este diagrama podemos ver todos los ficheros que tenemos que desarrollar para poder realizar el desarrollo de una funcionalidad, en nuestra arquitectura, lo primero tenemos que darnos cuenta es que tenemos dos entornos diferenciados en el desarrollo, tanto con el lenguaje como los framework.

Navegador:

En esta parte vamos a crear los ficheros que van a manejar la lógica de presentación, y el estado de nuestra aplicación (aparte trasladaremos parte del código que había en los controladores en los MVC clásicos en el servidor), ahora en el navegador, vamos a tener toda la lógica de presentación por lo cual esto va a provocar que tengamos muchísimas líneas de código en esta parte de la aplicación, por eso vamos a utilizar framework. A continuación, vamos a ver los ficheros que vamos a crear o modificar.

- ***Componente***: Es el encargado de controlar la región de pantalla de la funcionalidad, además integra la vista con los servicios y el modelo de los datos de pantalla.
- ***Plantilla***: Es la representación de la región de pantalla asociada al controlador (Se parece a los JSP de java.
- ***Modelo***: Es un objeto de datos que nos va a representar la información que se representa en la pantalla.
- ***Servicios***: Un servicio de angular será nuestra fachada con los servicios en servidor.
- ***Mapeo de las rutas***: Asocia una url a una pantalla, que tendremos asociado a un componente que se encargará de pintar la pantalla.

Servicio:

El servidor va a ser el responsable, tanto de la lógica de negocio como de la persistencia (Que no es poca cosa). También será el responsable de la seguridad.

- ***Controlador***: Lo vamos a utilizar para crear los métodos rest que centralizan la comunicación con el servidor.
- ***Datos***: Son objeto bean, que modelizan la información de nuestra información que vamos a mandar.
- ***Servicio***:Donde vamos a escribir la lógica de negocio. Estos servicios nos van a servir de fachada (para agrupar todo) y las transacciones van a empezar y terminar en un método del servicio.
- ***Dao***: Esta clase será la responsable del acceso a los datos.
- ***Entidad***: Es el bean en el que se modela en el ORM la persistencia de la funcionalidad.

Conclusión

En una primera impresión nos puede parecer que tenemos un montón de ficheros para realizar una aplicación, pero tenemos que tener en cuenta que en este tfc estamos detallando cómo montar un proyecto empresarial, en los cuales por ejemplo en un erp podemos tener muchas decenas de pantallas y cientos de miles de líneas de código, por lo cual nuestra principal preocupación es la de que el código sea mantenible frente a reducir el número de componentes.

Por otro lado también tenemos que tener en cuenta que en los últimos años hemos pasado de desarrollar aplicaciones, en las que la responsabilidad de pintar la pantalla estaba en el

servidor, debido a que http es un protocolo sin estado hacía un modelo más complicado realizar aplicaciones con estado o ria. Ahora debido a la evolución de los navegadores y los motores de javascript que al menos en mi experiencia personal empezó sobre el 2006 con jquery, firefox que nos empezó a permitir que el navegador empezará asumir más responsabilidad y con la aparición en 2008 de chrome y el motor v8. En resumen, ahora una aplicación en el navegador ya puede asumir muchas más responsabilidades, lo cual nos ha permitido sacar la responsabilidad de pintar la pantalla, como de la lógica de presentación del servidor, y además nos permite realizar aplicaciones ricas con estado de forma mucho más cómoda que en el servidor.

Por qué hemos decido utilizar 2 lenguajes de programación como javascript 6 + typescript y java8.

javscript + typescript

La razón de usar javascript, no es otra que para poder trabajar en el navegador no tenemos otro lenguaje de programación, la razón de usar typescript es porque por un lado los navegadores a dia de hoy no implementan javascript 6 (las clases y los módulos...), por otro lado nos permite usar tipos estáticos, que para una aplicación empresarial es algo fundamental y además la gente de angular 2 (google) lo recomiendan para utilizar su framework siendo una herramienta typescript de microsoft..

Java + spring + hibernate

Aqui tenemos muchas más opciones, ahora por ejemplo se ha puesto de moda la pila MEAN (nodejs + express + mongodb), y podríamos pensar por qué no utilizar esta solución para una solución empresarial.

> *En nuestro caso tenemos que ver que es una aplicación empresarial, que se va a encargar de gestionar la contabilidad de una empresa (a fin de cuentas, dinerito).*

Voy a explicar un poco por encima las características de una aplicación MEAN y sus ventajas:

- Gestiona las peticiones de una manera mucho más rápida.
- Está todo escrito en el mismo lenguaje de programación, por lo cual le permite tener librerías isomórficas que pueden trabajar en el servidor o en el navegador según las necesidades (ejemplo REACT). Esto es importante para las aplicaciones móviles.

- Y podemos persistir el modelo que tenemos en la capa de presentación en la persistencia.
- Las peticiones no son bloqueantes por lo cual son mucho más rápido.

En algunos artículos, blogs... se habla que no hay xml (pero lo que trabajamos en framework modernos java hace mucho que salvo el web.xml no se crea ningún fichero xml). Y con java 8 tenemos ciertos guiños a la programación funcional, como la lambda, stream, clousure. Y JavaScript 6 se ha dado un salto muy fuerte a cómo se trabaja en java y c# (como las clases, los ámbitos, un sistema modular) (al final los lenguajes tienden a converger).

Leyendo todo esto, podemos decir como no montamos la parte servidora en un proyecto empresarial con el MEAN. Pues que para conseguir todo la anterior tiene ciertas características o condicionantes.

Podríamos decir que la pila MEAN, para conseguir todo lo anterior se basa en que las operaciones no son bloqueantes, para conseguir esto cómo lo conseguimos:

- No podemos hacer operaciones con gran peso de procesamiento.
- Las operaciones de a la capa de persistencia no son bloqueantes, es decir se registra la operación guarda, pero no se espera a que se guarden los datos. (por lo cual puede darse en algún caso que perdamos los datos)

Esto en algunos escenarios es deseable, pero en un sistema empresarial, necesitamos garantizar la integridad de los datos por encima de todas las cosas, ya que los datos es lo más importante. Esta premisa necesitamos cumplir los siguientes requisitos:

- Por un lado, necesitamos que cuando nos devuelva el control, saber si han guardado o no correctamente los datos.
- Además, la integridad de los datos, por ejemplo, si tenemos un asiento con dos líneas, sólo será válido, si se ha guardado la cabecera y las dos líneas del asiento (O todo o nada) Es decir que si de los 3 registros ha fallado uno no se deben guardar los 3.

Estas dos características las podemos resolver con las transacciones, teniendo en cuenta que las operaciones en una B.D de datos es uno de los puntos en los que nuestra operación va invertir más tiempo, ya que hasta que el sistema de B.D no ha guardado los datos no nos va devolver el control.

En un sistema empresarial el sistema tiene que ser capaz en algunos de realizar muchos cálculos o operaciones pesadas de proceso.

Estas dos limitaciones anteriores no casan muy bien con la definición del MEAN.

Una vez que hemos visto una alternativa y porque para un sistema empresarial clásico sus limitaciones no nos valen, vamos a ver que nos ofrece Spring que es la solución que hemos elegido.

- Desarrollo ligero y muy poco invasivo, con objetos Java de clase simple antiguos (POJO).
- Acoplamiento débil mediante la inyección de dependencias y la orientación de interfaz.
- Programación declarativa mediante aspectos y convenciones comunes.
- Reducción del código reutilizable mediante aspectos y plantillas.
- Es un marco de trabajo que nos va a proporcionar, transacciones, integración con la capa de persistencia tanto JDBC, como un ORM... y otras posibles alternativas.

Funcionalidad horizontal

Esta es la funcionalidad, que vamos a realizar que va afectar a todas las operativas, en nuestro caso utilizaremos los siguientes pasos:

- Utilizar en spring la AOP.
  - El sistema de los logs.
- Utilizar mecánicas de la JDK.
  - Tipos genéricos.
    - Un dao y service genérico.
  - ....
- En angular 2 usaremos:
  - Componentes comunes.
    - El componente para esperar de una librería
    - Un pequeño grid.
  - Utilizar las librerías del framework.
    - Mecanismos de la ventana modal.
- Particularizaciones de los framework
  - Spring
    - Configuración del sistema transaccional.
    - Configuración del módulo de seguridad.
    - Configuración de los contenidos estáticos
    - Integración entre angular 2 y spring.
  - Angular 2

    - Configurar el enrutador para que utilice la #, y así evitamos un error al dar al F5.
- ....

Arquitectura de la parte front-end

Vamos a montar una aplicación SPI, por lo cual nos vamos apoyar en el framework angular 2, para dirigirlo todo más rxjs.

SPA (Simple-page application)

Es una aplicación de página única es una aplicación web o es un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio. En un SPA todos los códigos de HTML, JavaScript y CSS se cada de una vez o los recursos necesarios se cargan dinámicamente como lo requiera la página y se van agregando, normalmente como respuesta de las acciones del usuario. La página no tiene que cargar otra vez en ningún punto del proceso tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación. Las interacciones con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está detrás.

Angular 2

Es un marco de trabajo para crear aplicaciones ria en el servidor, que nos proporciona las siguientes características:

- Velocidad: Calcula las actualizaciones basadas en los cambios en los datos y no en el DOM, para que sean más rápidas y podamos escalar a conjuntos de datos más grandes.
- Móvil: Se coordina con el servidor y Web Workers, para conseguir una sensación suave, esto nos ayudará en el desarrollo móvil.
- Flexible: Por un lado, soporta varios lenguajes, JavaScript, TypeScript y Dart. Admite configuraciones de componentes con objetos y decoradores (son como las anotaciones de java) y programación funcional reactiva, con flujos de datos unidireccionales, soporte para objetos observables y estructura de datos inmutables. (para lo cual se va apoyar en la librería RxJS

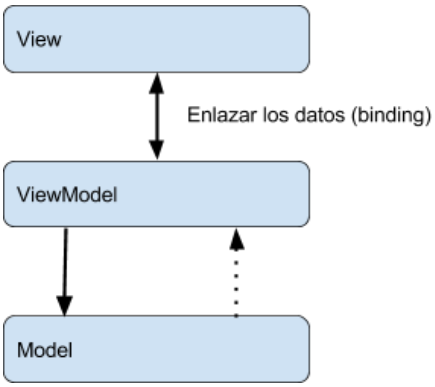
Arquitectura Model–view–viewmodel

Cuando hablamos del MVVM, se hace referencia a cómo tenemos que estructurar nuestra aplicación, en el cliente. El patrón MVVN se diseñó para la programación orientada a eventos y además nos proporciona una separación entre lógica de presentación y la lógica de negocio.

El MVVM se compone:

- ***Model (modelo)***: El modelo representa el estado, de los datos.
- ***View (vista)***: La vista es la interfaz de usuario.
- ***View Model ()***: Es una abstracción de la vista en la que se expondrán las propiedades públicas y métodos.
- ***Binder (enlazar)***: Es vinculación de los datos de la vista con el modelo, que se realizará de forma automática (sincronizando el estado con lo que vemos en pantalla).

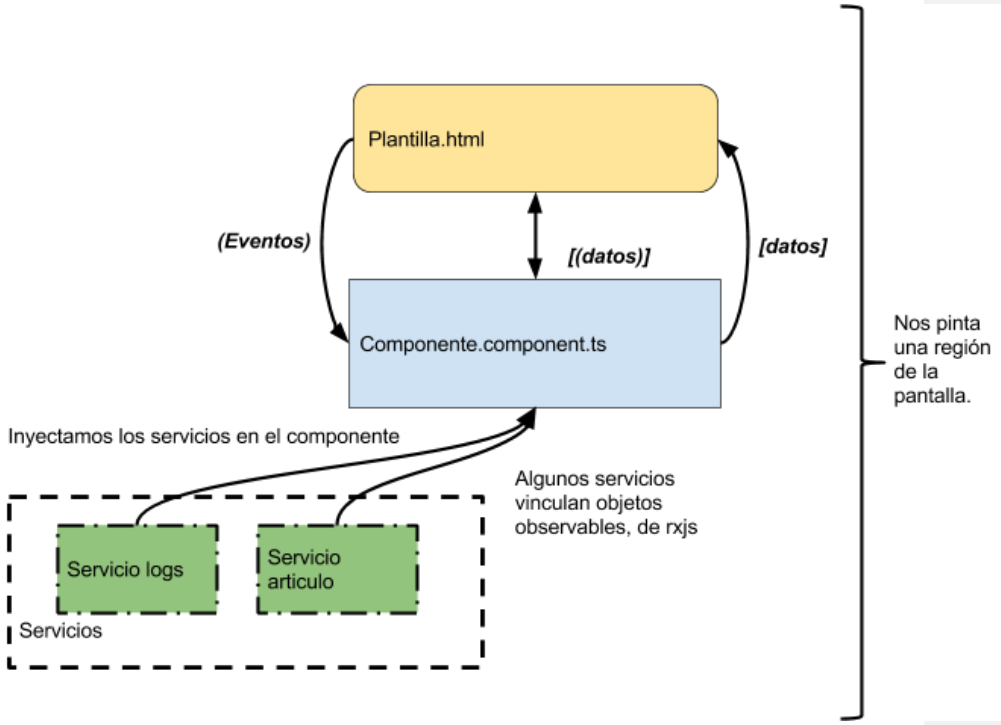
En la siguiente figura podemos ver una representación teórica del patrón.



Este patrón se creó para separar la lógica de presentación, de la lógica de negocio.

Arquitectura de angular 2

En la siguiente figura podemos observar de forma genérica como trabaja el framework para pintar una región de pantalla:

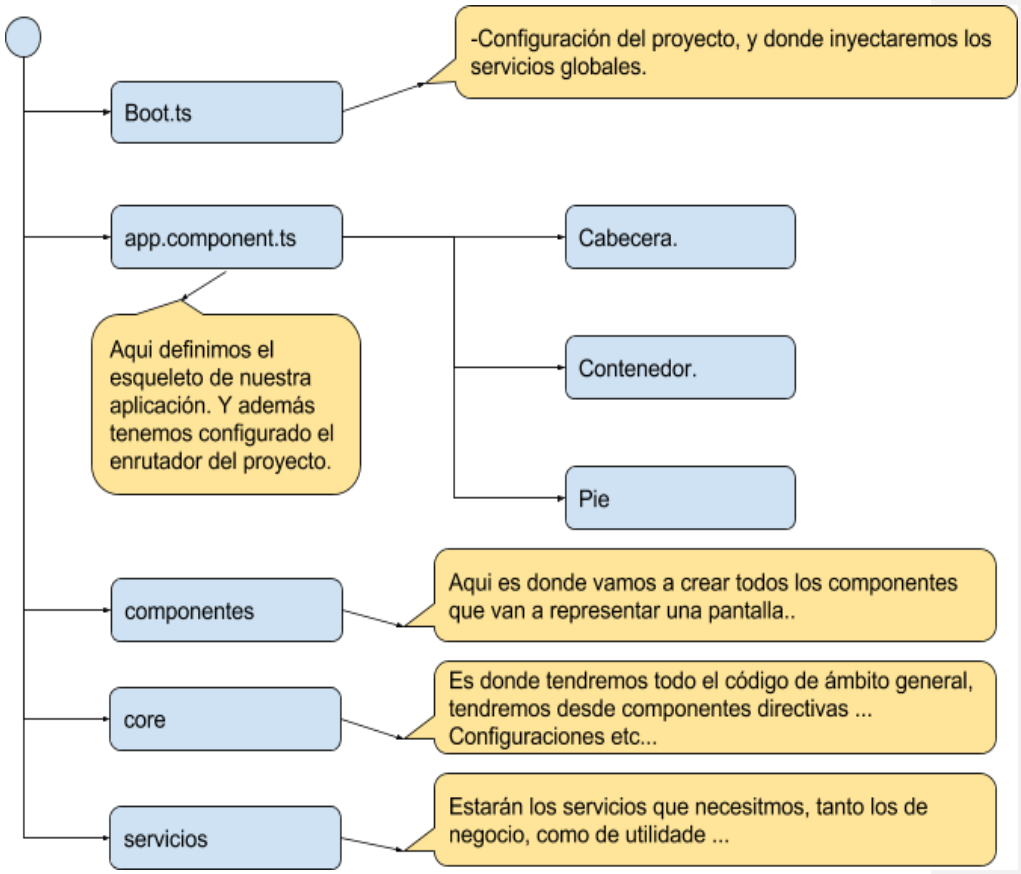


Como podemos ver en el diagrama anterior, para pintar cada una de pantallas nos apoyaremos en los siguientes elementos generales:

- ***Componente***: Es la clase que controla una región de la pantalla. Digamos que es la que engancha todo desde la plantilla donde vamos a indicar cómo maquetar la pantalla hasta los diferentes servicios.
- ***Plantilla***: Es la que representa la región que vamos a pintar en pantalla.
- ***Servicios***: Es el encargado de acceder al servidor para recuperar, los datos, guardarlos y enviarlos.

Organización de un proyecto en angular 2:

En la siguiente figura podemos ver cómo se va a organizar el proyecto en la parte de cliente:

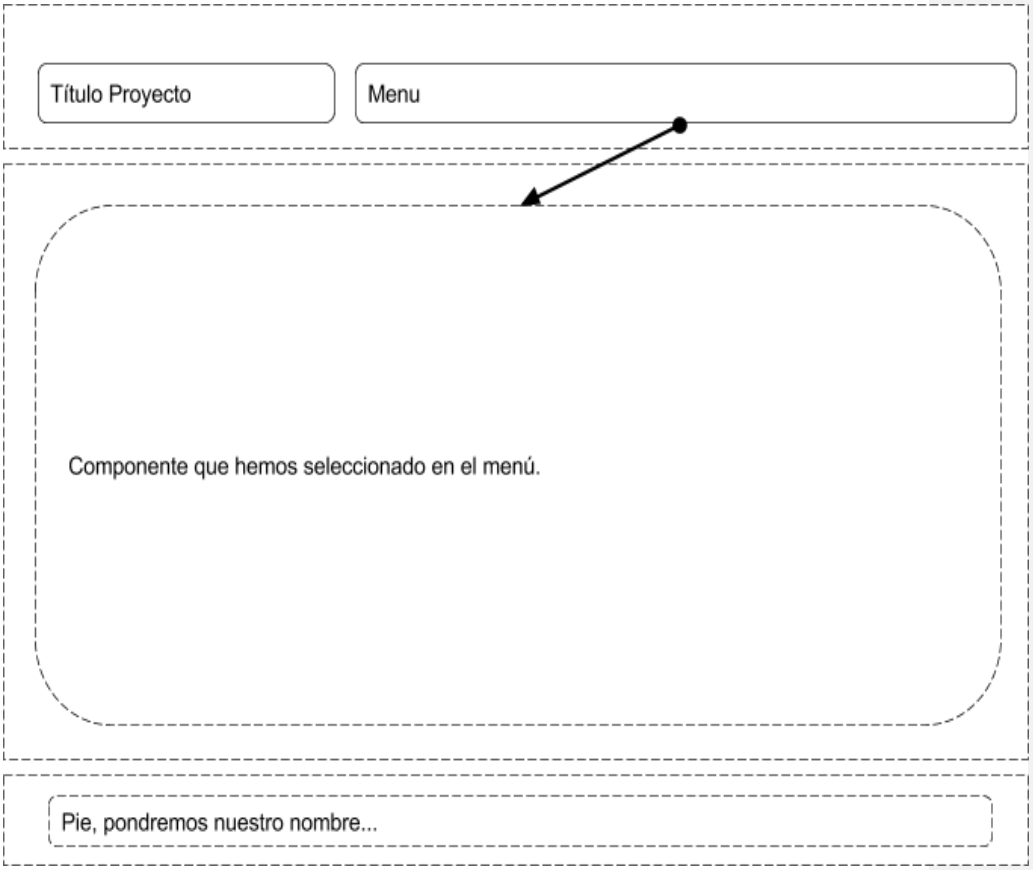


De la figura anterior lo dos ficheros más importantes, serán el boot.ts que es donde se inicializará el proyecto y el app.component.ts que es el componente raíz de toda nuestra aplicación SPA.

Esqueleto de nuestra aplicación

En este apartado vamos a definir cómo se va organizar la pantalla, para una aplicación multipantalla como la nuestra

En la siguiente figura podemos ver el esqueleto de la pantalla:



Nuestra aplicación se va a dividir en 3 secciones que vamos a detallar a continuación.

- ***Cabecera***: Tendremos la información referente al título del proyecto y un menú de navegación en el que tendremos acceso a las diferentes pantallas de nuestra aplicación. (cabecera.component.ts)
- ***Cuerpo***: Este es un contenedor, donde cargaremos cada una de las pantallas.

- ***Pie***: Aquí tendremos la información del proyecto que consideremos relevante. (pie.compoent.ts)

Esta organización de pantalla está ligada directamente con el modulo del router.

Modulo

Las aplicación será modular, para lo cual utilizaremos el mecanismo que se ha creado en el estándar ES2015, para módulos de la nueva especificación de JavaScript.

Definición*: Un módulo en general es un bloque coherente de código dedicado a un solo propósito.*

En angular tenemos diferentes tipos de módulos:

- ***Componentes***: Es un módulo donde definiremos los componentes siguiente la siguiente sintaxis .component.ts.
- ***Servicios***: Es donde definiremos los servicios de nuestra aplicación.
- ***Biblioteca***: Es un módulo que agrupa varios módulos privados, en los que nos hace de fachada. Ejemplo angular/core. Así es como está agrupado todo el framework de angular 2.
- ***Variables***: Es donde guardaremos las variables o constantes. Lo usaremos para parametrizar nuestra aplicación o crear constantes, enumerados ...
- ***Funciones***: Es donde guardaremos las funciones de utilidades que necesitemos sobre nuestro desarrollo.

Ahora que ya hemos visto que es un módulo y algunos de los diferentes tipos de módulos que vamos a tener en una aplicación de JavaScript, vamos a ver cómo organizar una aplicación estándar.

1. ***boot.ts***: Este es el modulo que va inicializar la aplicación, configurar el sistema de enrutamiento y inyectar todos los módulos que nos interesen que vea la aplicación.

boot.ts

```
import 'rxjs/operator/map';
import 'rxjs/operator/mergeMap';
import 'rxjs/observable/interval';

import {bootstrap} from 'angular2/platform/browser';
import {ROUTER_PROVIDERS, LocationStrategy, HashLocationStrategy} from 'angu-
lar2/router';
import {provide} from 'angular2/core';
import {HTTP_PROVIDERS} from 'angular2/http';
import {AppComponent} from './app.component';
import {MenuService} from './services/menu/menu.service';
import {AsientoService} from './services/contabilidad/asiento/asiento.service';
import {DbpDialogo} from './core/modal/dialogo';

bootstrap(AppComponent,[
   ROUTER_PROVIDERS // Provedor de enrutamiento
   ,HTTP_PROVIDERS // Provedor del recurso http
   ,MenuService,AsientoService
   ,DbpDialogo
   ,provide(LocationStrategy, {useClass: HashLocationStrategy})
 ]);
```

Es el boot que hemos utilizado para nuestra aplicación, aquí le indicamos como tiene que funcionar el enrutador, y le inyectamos los diferentes servicios que vamos a utilizar.

2. ***app.component.ts***: Es el componente base de toda la aplicación, el primero que se carga y desde el que vamos a definir el esqueleto de la aplicación.

app.component.ts

```
import {Component} from 'angular2/core';
import {ROUTER_PROVIDERS,RouterOutlet,RouteConfig} from 'angular2/router';
import {Inicio} from './components/inicio/inicio.component';
import {About} from './components/about/about.component';
```

```
import {Contacto} from './components/contacto/contacto.component';
import {CabeceraComponent} from './components/comun/cabecera.component';
import {PieComponent} from './components/comun/pie.component';
import {AsientoComponent} from './components/contabilidad/asientos/asiento.component';
import {ModalComponent} from './components/ejemplos/modal/modal.component';
@RouteConfig([
  {path: '/Inicio',   component: Inicio,    as: 'Inicio'},
  {path: '/About',    component: About,     as: 'About'},
  {path: '/Contacto', component: Contacto,  as: 'Contacto'},
  {path: '/Asiento',  component:AsientoComponent, as:'Asiento'},
  {path: '/ModalEjemplo', component: ModalComponent, as:'ModalEjemplo'}
])
@Component({
    selector: 'my-app',
    templateUrl: '/src/app/app.component.html',
    directives:[CabeceraComponent,RouterOutlet,PieComponent]
})
export class AppComponent {}
```

Es el componente raíz de nuestra aplicación, como podemos ver por un lado tenemos la configuración del enrutador, y por otro el esqueleto de la aplicación con la cabecera, contenedor y pie.

> *Hemos explicado primero que son y cómo funcionan los módulos en JavaScript, porque aunque no es algo que nos proporciona el framework de angular 2, es una nueva característica de JavaScript 6, que vamos a utilizar tanto para poder organizar el proyecto, como crear las diferentes partes.*

Componente

Es una clase que controla una región de la pantalla a la que llamaremos vista. Todas las regiones de pantalla tienen asociado un componente.

Se crean, actualizar y destruyen componentes. Los cuales tienen un ciclo de vida en el que el programador puede actuar si es necesario.

A continuación, vamos a ver un ejemplo de componente.

```
@Component({
    selector: 'my-app',
    templateUrl: 'Hola caracola',

})
export class AppComponent {}
```

*Este componente de ejemplo es muy simple, solo escribe en la región de pantalla que controla '**Hola caracola**'*

Plantilla

La plantilla, digamos que representa la región de pantalla que representa a un controlador.

Haciendo un símil al mundo de java, se parece a un JSP (salvando las distancias).

Las plantillas en angular tienen una sintaxis, con la que tendremos condicionales, bucles, bindeos y el controlador angular es capaz de pintar la región de pantalla correspondiente.

Comentado [1]: Queda pendiente de ver como redactarlo mejor.

Ojo en la plantilla indicaremos los bindeos que pueden ser unidireccionales o bidireccionales, de cada componente.

Por ejemplo, podemos bindear una propiedad en la plantilla a un atributo de la clase que representa el controlador, si el controlador cambia el valor se cambiará en la plantilla. O otro caso distinto puede ser bindear un evento como el de click en la plantilla y asociarlo a una operación de la clase, cuando se ejecute el evento se ejecuta la operación.

Metadatos

Los metadatos, son los que le dicen a angular como procesar una clase.

Como ejemplo vamos a ver como se define un componente.

```
@Component({
  selector:    'tfc-selector',
  templateUrl: 'app/tfc.component.html',
  directives:  [TfcDbpComponent],
  providers:   [TfcService]
})
export class HeroesComponent { ... }
```

- **selector**: Es un selector css, donde pondremos el componente.
- **tempalteUrl:** Es la dirección de la plantilla, del controlador.
- **directives**: Es un array con los componentes y diferentes directivas que requiere la plantilla.
- **providers**: Es un array con los proveedores de inyección de dependencias.

La plantilla, los metadatos y el componente describen la vista.

Enlaces (Binding)

Es el mecanismo para actualizar los datos de manera automática y enlazar los eventos a acciones del controlador...

De esta manera por un lado nos quitamos de realizar esta tediosa tarea manual (Nos ahorramos tiempo de desarrollo, como facilita el mantenimiento) y lo que es más importante evitamos errores.

En angular los bindeos afectarán a:

- ***Datos***: Que es el modelo que vamos a presentar en la plantilla y cuando se cambia un dato que cambie en la pantalla.
- ***Eventos***: Son las acciones que se lanzan desde la pantalla, que propagamos al controlador.

Una vez que sabemos, que vamos a enlazar, vamos a ver las direcciones.

Ahora en la forma de declararlo nos indica que dirección va a ser el enlace:

- **[**<---**]**: Esto indica que la acción empieza en el componente a la plantilla (pantalla). Ejemplo vamos a actualizar un dato.

```
[elemento]="elementoSeleccionado"  // Llama a la función de elemento seleccionado y carga en el elemento.
```

```
{{elemento.nombre}} // Pinta el atributo nombre del modelo elemento.
```

Este último ejemplo, hace uso de la interpolación (se parece al EL de jsp), para realizar el bindeo y es el único caso en el que no queda representado en la sintaxis la dirección.

- **(**--->**)**: Esto indica que la acción empieza en la pantalla y llega al controlador.

```
<div (click)="selecionarDato(dato)"></div>
```

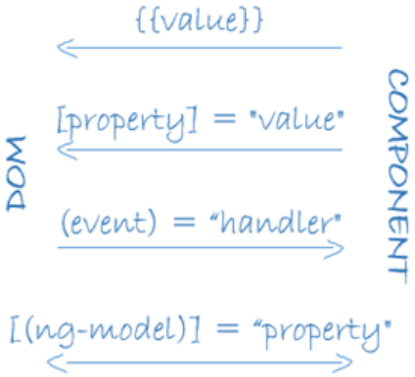
Al realizar u*n click sobre la región que representa el div, se llamará al método de selecionarDato(dato). En este caso la acción la provoca un evento de pantalla.*

- **[(**<---->**)]**: Esto indica que la acción empieza en la pantalla y llega al controlador y viceversa.

```
<input [(ngModel)]="modelo.nombre"/>
```

En este ejemplo lo que hacemos es enlazar la propiedad modelo.nombre, esto quiero decir que si el controlador cambia la propiedad se cambia en la pantalla y si el cambio se realizar en la pantalla, re repercute en el modelo que tenemos en el controlador.

En la documentación de angular lo resumen muy bien esto en la siguiente figura.



Directivas:

Son las encargadas de dar las indicaciones en la plantilla, para transformar el árbol dom.

> *Los componentes, también son una directiva, pero debido a su importancia en el framework, se le decidió dar más peso.*

Aparte de los componentes, tenemos dos tipos de directivas:

- ***Atributo***: Se llama asi por que aparecen dentro de una etiqueta como si fuera un atributo. Se pueden usar para alterar la apariencia, el comportamiento, enlazar datos.

```
<input [(ngModel)]="modelo.nombre">
```

> *En este ejemplo enlazamos unos datos.*

- **estructurales**: Alteran el diseño añadiendo, quitando o sustituyendo elemento en el dom.

```
<div *ngFor="elemento of elementos"></div>
<tfc-detalles *ngIf="elementoSelecionado"></tfc-detalles>
```

1. **ngFor: Pintar tantos div como elementos tengamos.*
2. **ngIf: Si existe el elemento seleccionado se pinta el tfc-detalles.*

Servicios:

Un servicio es una clase que debe hacer algo específico y hacerlo bien.

Las aplicaciones de angular 2, serán grandes consumidoras de servicios y digamos que depende del desarrollador los diferentes tipos de servicios que vamos a tener, a continuación, vamos a ver algunos:

- Puede ser de tipo back-end: Se enganchará, con el servidor, por ejemplo, tendremos un servicio por cada entidad. (Pueden ir emparejados con los del servidor y llamarse igual si puede ser para facilitar el mantenimiento).
- Puede ser utilidades: Como por ejemplo podemos crear unas utilidades para centralizar los logs, que ahora por ejemplo usamos la consola, pero si mañana aparece un framework estilo logback (en java), solo tenemos que cambiar el código de la utilidad.
- Pueden ser configuraciones: Cualquier tipo de configuración, el idioma, el formato de fecha, (lo que necesitemos ...)
- Cualquier cosa que necesitemos.

Inyección de dependencias:

Es una manera de proporcionar nuevas instancias de una clase con las dependencias totalmente formadas que requiere.

> *La mayoría de las dependencias serán servicios.*

La inyección de dependencias va a utilizar dos mecanismos, con diferentes responsabilidades:

- ***Inyector*** (injector): Es el encargado de crear la instancia, si no existe (utilizara al proveedor que tenga registrado), y mantenerla en el contexto. Si existe una instancia la devuelve.
- ***Proveedor*** (provider): Es el que sabe cómo crear la instancia de la clase que vamos a inyectar.

Es muy importante saber dónde definamos la inyección de dependencias, por ejemplo:
1. Si lo hacemos en el bootstraap, será la misma instancia para toda la aplicación.
2. Si lo hacemos en un componente, cada vez que se cree ese componente se crear una nueva instancia.

> *Esto que parece trivial es importante, ya que nos indicara en qué contexto la clase se comportara como un Singleton etc...*

Herramientas de angular 2

Ahora vamos a ver otros tipos de herramientas que nos proporciona angular y una breve descripción.

Animaciones

Nos proporciona una biblioteca de animaciones, que nos abstrae de la css.

> *No lo vamos a usar, y tampoco en la beta de angular 2 no lo tienen documentado ni la documentación de la librería.*

Bootstrap

Se encarga de arrancar angular y establecer las configuraciones.

> *En angular 2 el index.html no se procesa para evitar choques con el servidor y por razones de seguridad, esto nos permitirá usar en el servidor en el index {{}}*

El bootstrap lo tenemos que invocar con el componente de aplicación y se realizarán las siguientes acciones.

1. Buscar en el árbol DOM, el selector del componente de aplicación, que es la región de pantalla que va a controlar angular 2.
2. Se crear un nuevo inyector (desde el de la plataforma). Opcionalmente puede anular el comportamiento por defecto de la configuración, invocan en el bootstrap el `componentInjectableBindings` .
3. Se crea una zona de angular y la conecta a la instancia de detección de cambios de angular.
4. Crear un DOM emulado en la sombra, en la raíz del elemento relacionado y se carga la plantilla.
5. Se crea una instancia del componente especificado.
6. Por último, asociado la detección de cambios a los proveedores de datos.

Comentado [2]: OJO, tenemos que revisar esta ultimo paso por que no queda claro en la documentación.

La zona en angular 2, es la región de pantalla donde se verán afectados los cambios entre la pantalla y el modelo. (para más información revisar en el api ngZone).

En este apartado vamos a inyectar los servicios comunes y las posibles configuraciones.

```
@Component({selector: 'my-app', template: 'Hello {{ name }}!'})
class MyApp {
 name: string = 'World';
}

function main() {
 return bootstrap(MyApp);
}
```

A continuación, vamos a ver un ejemplo, de cómo cambiar el enrutador de angular 2 para que ponga en las rutas parciales la #. Inyectando la clase que va a establecer la estrategia del location.

```
@Component({selector: 'my-app', template: 'Hello {{ name }}!'})
class MyApp {
 name: string = 'World';
}

function main() {
 return bootstrap(MyApp,ROUTER_PROVIDERS
,provide(LocationStrategy, {useClass: HashLocationStrategy}));
}
```

El orden de poner el router_providres y con la estrategia importa ya que el router providers establecer una por defecto y el segundo la reemplaza la estrategia por defecto.

Detección de cambios

Angular 2 nos proporcionara ciertos mecanismos, para controlar la detección de cambios, pararlos... Lo cual nos permitirá por ejemplo si necesitamos mejorar los tiempos porque tenemos muchos tiempos decidir cuándo se hace la detección de los cambios.

Por un lado, tendremos, la directiva ngModel, que es la encargada, de realizar el bindeo con los atributos, en el apartado de formulario, lo vamos a ver su funcionamiento más detallado.

Por otro lado, angular 2 nos va a proporcionar mecanismos para poder desactivar o gestionar la detección de cambios aparte algunos enganches del ciclo de vida de una directiva van asociada a la detección de cambios.

Enrutador (Router)

Es una herramienta que nos permite navegar en una aplicación multipantalla.

Una URL en la barra del navegador y entramos en una pantalla, este es el modelo que sigue el router de angular.

Para trabajar con el router de angular tenemos que tener en cuenta 3 cosas:

- El mapeo de la ruta a un componente @RouterConfig.
- El mecanismo para indicarle la ruta `[routerLink]`. O cambiando la url en el navegador.
- Indicarle donde vamos a pintar el componente, `<router-outlet>`

@RouterConfig

Usaremos este decorador, para mapear una ruta a un componente, se apoya en la clase ***RouteDefinition***. Por cada mapeo vamos a configurar las siguientes propiedades:

- **path**: Es la ruta a la que vamos a asociarlo.
- **name**: Es el nombre de la ruta, la cual tiene que estar escrita en PascalCase
- **component**: Es el componente que vamos a utilizar

Comentado [3]: Explicar que es el PascalCase

```
@RouterConfig([
 {path:'/asiento', name:'Asiento', component:AsientoComponent},
 {path:'/inicio', name:'Inicio', component:InicioComponent}
])
```

routerLink

Es la directiva de atributo (que vamos a vincular a una etiqueta **a**), que vamos a utilizar, para indicarle que mapeo tiene que seguir, se guiará por el atributo name.

```
<a [routerLink]="['Asiento']"> asiento </a>
```

router-outlet

Es simplemente la etiqueta donde vamos a cargar el componente

```
<router-outle></router-outlet>
```

El router de angular necesita saber cuál es la base con la que vamos a trabajar para lo cual utilizaremos un metadato de HTML 5 de la cabecera que es la base <base href="/" />.

Podemos también desde el código enlazar una operación del enrutador utilizando la clase router, navigation.

Comentado [4]: En el tema de enrutamiento de angular podemos desarrollarlo a un mas. Esto es la forma básica de trabajar.

Para que el enrutador funcione es necesario que, en la carga inicial del proyecto, activemos el módulo inyectando sus providers, una vez establecido el enrutador, le vamos a indicar que los mapeos se ponga una # para que cuando se dé al F5 se mantenga en la misma página (usaremos ***HashLocationStrategy***).

```
import {ROUTER_PROVIDERS, LocationStrategy, HashLocationStrategy} from 'angular2/router';

....
```

```
bootstrap(AppComponent,[
    ROUTER_PROVIDERS // Provedor de enrutamiento
    ,provide(LocationStrategy, {useClass: HashLocationStrategy})
  ]);
```

Eventos

Al igual que en el DOM se utilizan los eventos, el mecanismo de comunicarse entre diferentes servicios y componentes usaremos los objetos observables de RxJs

Formularios

Nos ofrece ciertas herramientas para trabajar con los formularios, con diferentes escenarios de entrada de datos, validaciones etc...

Nos da soporte para realizar las siguientes operaciones:

- Enlace bidireccional de los datos.
- Control de los cambios.
- Validaciones
- Manejo de los errores

Para lo cual usaremos las siguientes directivas:

- ***ngModel***: Es la directiva encargada de proporcionar la bidireccional dad de los datos.
- ***ngControl:*** Es para el seguimiento de los cambios y validación.

Estado	Clase si es verdad	Clase si es falso
El control ha sido visitado	ng-touched	ng-untouched
El valor del control ha cambiado	ng-dirty	ng-pristine
El valor del control es válido	ng-valid	ng-invalid

- ***ngSubmit:***

Para crear un formulario tenemos que seguir los siguientes pasos:

1. Creamos la clase de modelo que represente, al formulario.

```
export class EjemploModel{
  constructor(public id:number,public descripcion:string){
```

```
  }
}
```

Creamos un modelo con dos campos uno es el Id, y el otro es una descripción.

2. Creamos el controlador encargado de gestionar el formulario

```
@Component({
  selector: 'ejemplo-form',
  templateUrl:'ejemplo-form.component.html'
})
export class EjemploFormComponent{
   modelo:EjemploModelo = new EjemploModelo(2,' una descripcion');
}
```

Creamos un controlador y le asociamos el modelo de datos, con unos datos de ejemplo creados.

3. Creamos la plantilla que va a representar el formulario.

```
<div class="container">
  <h1>Ejemplo Form</h1>
  <form>
    <div class="form-group">
      <label for="id">Id</label>
      <input type="text" class="form-control" required>
    </div>

    <div class="form-group">
      <label for="descripcion">descripcion</label>
      <input type="text" class="form-control">
    </div>
    <button type="submit" class="btn btn-default">Enviar</button>
  </form>
</div>
```

4. Agregamos la directiva ***ngModel*** en cada campo del formulario.

```
<input type="text" class="form-control" required [(ngModel)]=”modelo.id”>
```

```
<input type="text" class="form-control" [(ngModel)]=”modelo.descripcion”>
```

5. Agregamos la directiva ***ngControl*** en cada campo del formulario.

```
<input type="text" class="form-control" required [(ngModel)]=”modelo.id” ngCon-
trol=”id”>
```

```
<input type="text" class="form-control" [(ngModel)]=”modelo.descripcion” ngCon-
trol=”descripcion”>
```

En cada ngControl vamos a poner un nombre identificativo, aquí se ha seguido la pauta de poner los nombres de los atributos del modelo.

6. Añadir CSS personalizado, para proporcionar información visual. (Este apartado lo vamos a configurar una sola vez en la CSS de la aplicación).

```
.ng-valid[required] {
  border-left: 5px solid #42A948; /* green */
}
.ng-invalid {
  border-left: 5px solid #a94442; /* red */
}
```

7. Mostrar y ocultar los mensajes de error y validación.

```
<input type="text" class="form-control" required [(ngModel)]=”modelo.id” ngCon-
trol=”id” #id="ngForm">
<div  [hidden]=”id.valid” class=”alert alert-danger” >
  Id requerido.
</div>
```

Aquí lo que hacemos es pasarle el ngForm a la variable con el nombre id, para que fuera del controlador podamos acceder al valor del ngControl y saber si es válido o no.

8. Gestionar el envío del formulario usando la directiva ***ngSubmit***.

```
<form (ngSubmit)="onSubmit()" >
```

Al hacer submit, lo llamaremos al método submit.

9. Desactiva el botón de envío del formulario hasta que el formulario sea válido.

```
<form (ngSubmit)="onSubmit()" #ejemploForm="ngForm">
```

```
<button type="submit" class="btn btn-default"
    [disabled]="!ejemploForm.form.valid">Enviar</button>
```

Http

Es la herramienta que utilizaremos para comunicarnos con el servidor, nos permite obtener, modificar, guardar o eliminar los datos.

Para poder realizar esto implementa las siguientes operaciones concretas de http

- GET: Recuperación de un recurso.
- POST: Modificación de un recurso.
- PUT: Creación de un recurso.
- DELETE: Eliminación de un recurso.
- PATCH: Modificación parcial de un recurso.

Aparte de estos métodos de http, esta herramienta va a implementar las siguientes operaciones genéricas.

- request: Procesar cualquier recurso, es para un uso más general.
- parámetros cabecera: Nos permite cambiar las variables de cabecera de http.

Una vez que hemos visto que nos proporciona para http angular 2, vamos a ver qué pasos tenemos que seguir ya que es un módulo separado de angular 2 que tenemos que inicializar para poder usarlo.

En nuestro caso tiene una dependencia transitiva a la librería RxJS, por lo cual lo primero que vamos a ver es como solucionar esta dependencia.

1º) Tenemos que pasar la dependencia con systemjs al proyecto.

```
System.config({
              packages: {
                 ....
                 '../node_modules/rxjs': { defaultExtension: 'js' }
              },
              paths: {
                 'rxjs/observable/*' : '../node_modules/rxjs/add/observable/*.js',
                 'rxjs/operator/*': '../node_modules/rxjs/add/operator/*.js',
                 'rxjs/*': '../node_modules/rxjs/*.js'
              }
           });
```

2º) Ahora que ya lo hemos configurado en systemjs no es suficiente ahora lo tenemos que importar, que lo haremos en el boot, para que tenga acceso todas las clases de la aplicación.

```
import 'rxjs/operator/map';
import 'rxjs/operator/mergeMap';
import 'rxjs/observable/interval';
```

Ahora que ya tenemos la librería, vamos a configurar el módulo de http de angular.

1º) Importamos la js de la librería:

```
    <script src="../node_modules/angular2/bundles/http.dev.js"></script>
```

2º) Lo inyectamos en el inicio de angular 2, para que la pueda usar.

```
import {HTTP_PROVIDERS} from 'angular2/http';

....

bootstrap(AppComponent,[
    ,HTTP_PROVIDERS // Proveedor del recurso http
 ]);
```

Enganches al ciclo de vida

Los componentes de angular tienen un ciclo de vida, al cual el programador se puede enganchar si lo necesitamos.

Para lo cual usaremos los interfaces para los ciclos de vida.

Interfaz	**método**	**descripción**
onChanges	ngOnChanges	Cambios en la entrada y cambiar de valor
onInit	ngOnInit	Después de los primeros onChanges
doChecked	ngDoChecked	Detección de cambios indicados por el desarrollo
afterContentInit	ngAfterContentInit	Después de estar inicializado al componente.
afterContentChecked	ngAfterContentChecked	Después de cada chequeado al componente.
afterViewInput	ngAfterViewInput	Después de la inicialización del componente.
onDestroy	ngOnDestroy	Justo antes de que se destruya de la directiva.

Tuberías (pipes):

Son servicios para transformar los valores de la pantalla. Haciendo un símil con los JSP, sería las funciones que nos proporciona las tld de un contenedor de servlet).

A diferencia de las funciones de una tld de java, usarán las tuberías que la entrada de uno será la salida de otra función, en principio las tuberías serán sin estado y nos permiten poner parámetros. Por ejemplos si queremos pasar a mayúsculas y quitar los blancos, usamos el upper | trim. (El resultado del upper se lo pasa al trim).

Angular 2 nos proporcionan, ciertas tuberías para usar en las plantillas, pero nos permite crear las nuestras. También nos permite que una tubería sea con estado (aunque rompe la forma de trabajar).

> *Las usaremos para realizar transformaciones en general, formatear valores. Pero se pueden usar para cualquier cosa.*

Ejemplo:

```
{{ cumple| date:"dd/MM/yyyy" }}
```

En este caso, nos formatea un campo fecha en el formato que le hemos indicado.

```
{{ cumple| date | upper}}
```

En este caso lo formateamos y lo pasamos a mayúscula.

> *No es una forma habitual para trabajar con este tipo de utilidades, pero tampoco es difícil adaptarse una vez se entiende el concepto de tubería.*

Los cambios que hay entre la beta y la RC

Mientras se ha estado haciendo el proyecto debido a que angular 2 estaba en fase beta se han ido haciendo cambios por lo cual algunas cosas de la documentación han cambiado respecto al proyecto final.

En el siguiente enlace podemos ver todos los cambios que hemos tenido que realizar por cada cambio en la beta de angular (https://github.com/blancoparis-tfc/tfcContabilidad/issues?q=is%3Aissue+is%3Aclosed+label%3A%22COMP%3A+Migrar+angular+2%22) cada una de las tareas tiene los ficheros asociados.

Aun así, vamos a enumerar algunos de los cambios.

- Todos los import, han pasado de angular2 a @angular el prefijo.
- El router que hemos utilizado ha pasado a router-deprecated

RxJS

Es una biblioteca para crear programas asíncronos y basados en eventos utilizando secuencias observables y operadores basados en el estilo LINQ (filter, map....).

Trabaja con secuencias de datos que RxJS las va a presentar como secuencias observables. Este le permite a nuestro programa suscribirse para poder recibir notificaciones asíncronas, como la llegada de nuevos datos.

Rxjs trabaja sin problema con datos síncronos (iterables) y asíncronos como promeses (Observables).

	Retorno individual	Retorno múltiple
Pull/ síncrono / iterable	Object	Iterables (Array | set| map |Object)
Push/ asíncrono / reactivo	Promesa	Observable

Ejemplo:
Iterables, lo vamos a utilizar cuando trabajemos con datos en memoria.

```
obtenerDatosDeMemoria()
.filter(s => s!=null)
.map(s => `${s} transformar`)
.forEach( s=> console.log(`siguiente => ${s}`))
```

Observable, lo vamos a utilizar cuando trabajemos por ejemplo con datos recuperados de la red.

```
obtenerDatosDeLaRed()
.filter(s => s!=null)
.map(s => `${s} transformar`)
.subscribe( s=>  console.log(`siguiente => ${s}`))
```

Patrón observador (Publicar-subscribir)

Este patrón define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

Este patrón nos sirve además nos proporciona una bajo acoplamiento entre los diferentes objetos.

Objetivo

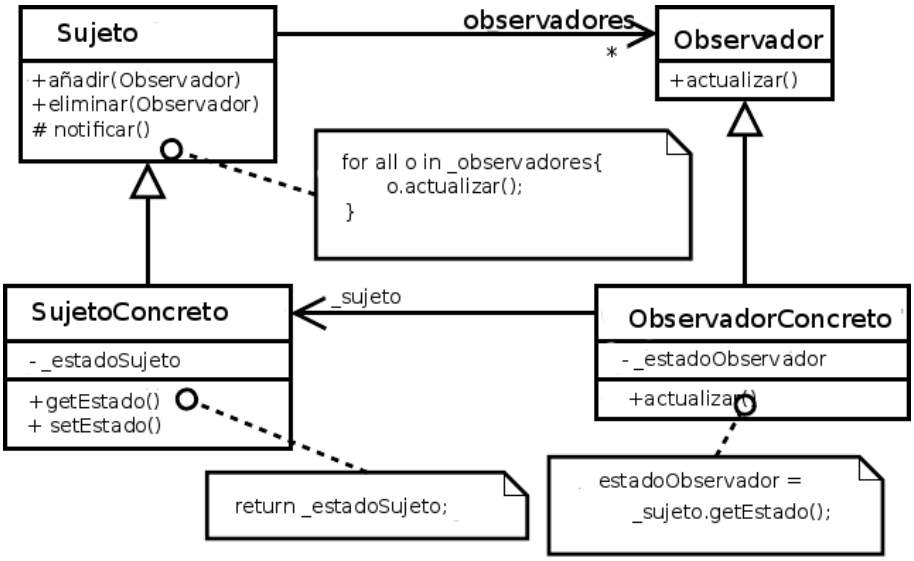
Definir una dependencia uno-a-muchos entre objetos, de tal forma que cuando el objeto cambie de estado, todos sus objetos dependientes sean notificador automáticamente. Se trata de desacoplar la clase de los objetos clientes de objeto, aumentando la modularidad del lenguaje, creando las mínimas dependencias y evitando bucles de actualización (espera activa o polling). En definitiva, normalmente, usaremos el patrón Observador cuando un elemento “quiere” estar pendiente de otro, sin tener que estar encuestando de forma permanente si éste ha cambiado o no.

Aplicabilidad

Donde podemos usar este patrón:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsula estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos. En otras palabras, cuando no queremos que estos objetos están fuertemente acoplados.

Estructura



Participantes

SUJETO:

- conoce a sus observadores. Un sujeto puede ser observado por cualquier número de objetos Observador.
- proporciona una interfaz para asignar y quitar objetos Observador.

OBSERVADOR

- define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.

SUJETOCONCRETO

- almacena el estado de interés para los objetos ObservadorConcreto.
- envía una notificación a sus observadores cuando cambia un estado.

OBSERVADORCONCRETO

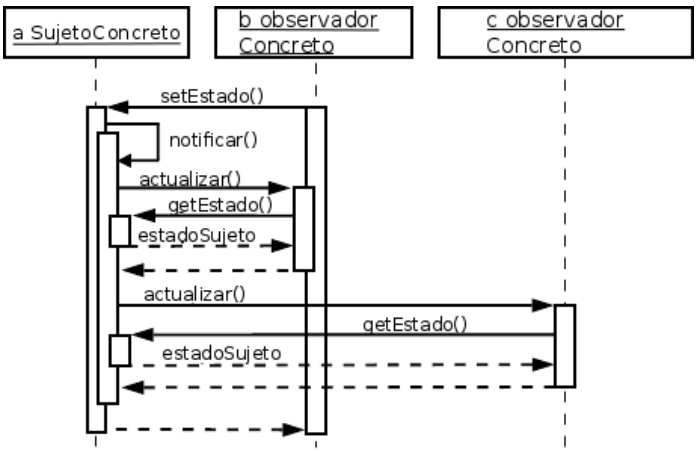
- mantiene una referencia a un objeto SujetoConcreto.
- guarda un estado que debería ser consistente con el del sujeto.
- implementa la interfaz de actualización del Observador para mantener su estado consistente con el del sujeto.

Colaboraciones

- El sujeto concreto notifica a sus observadores, cada vez que se produce un cambio que pudiera hacer que el estado de éstos fuera inconsistente con el suyo.

- Después de ser informado de un cambio en el sujeto concreto, un objeto ObservadorConcreto puede pedirle al sujeto más información. El ObservadorConcreto utiliza esa información usa esta información para sincronizar su estado con el del sujeto.

El siguiente diagrama de interacción muestra las colaboraciones entre un sujeto y dos observadores.



Empieza el proceso el objeto observador, pero podemos ver como espera a que le llegue la notificación para terminar el proceso de actualizar. Notificar no siempre es llamado por el sujeto puede ser llamado por otro.

Consecuencias:

- El patrón Observador permite modificar los sujetos y observadores de forma independiente. Es posible reutilizar objetos sin reutilizar sus observadores y viceversa.

- Acoplamiento abstracto entre Sujeto y Observador: Todo lo que un sujeto sabe es que tiene una lista de observadores, cada uno de los cuales se ajusta a una interfaz simple de la clase abstracta Observador. El sujeto no conoce la clase concreta de ningún observador. Por tanto, el acoplamiento entre sujetos y observadores es mínimo.

  Gracias a que Sujeto y Observador no están fuertemente acoplado, pueden pertenecer a diferentes capas de abstracción de un sistema. Un sujeto de bajo nivel puede

comunicarse e informar a un observador de más alto nivel, manteniendo de este modo intacta manteniendo de este modo la estructura de capas del sistema intacta.

- Capacidad de comunicación mediante difusión: A diferencia de una petición ordinaria, la notificación enviada por un sujeto no necesita especificar su receptor. La notificación se envía automáticamente a todos los objetos interesados que se hayan suscrito a ella. Al sujeto no le importa cuántos objetos interesados haya; su única responsabilidad es notificar a sus observadores. Esto nos da la libertad de añadir y quitar observadores en cualquier momento. Se deja al observador manejar u obviar una notificación.
- Actualizaciones inesperadas: Debido a que el observador desconoce al resto de observadores, no sabe las repercusiones que puede tener actualizar el estado de un sujeto etc...

Directrices de diseño de RxJS

Son unas directrices para trabajar con RxJS, que nos recomienda el grupo de desarrollo de RxJS. Esto permite añadir observadores sin modificar el sujeto u otros observadores.

> *Son recomendaciones, pero ni son de obligado cumplimiento, ni una verdad absoluta, pero se recomienda usarla siempre que se adapten a nuestro problema.*

Flujos asíncronos:

El típico evento de click ya en sí es un flujo asíncrono, el cual puede ser observado.

> *Se pueden crear flujos de datos de cualquier cosa como (variables, propiedades, caches, entradas de datos ...)*

En resumen, un flujo asíncrono se basa en la idea, ***que se escucha una corrientes y se reacciona en consecuencia***.

> *Flujo, en muchos sitios le llaman stream, que es la palabra técnica que se usa en inglés.*

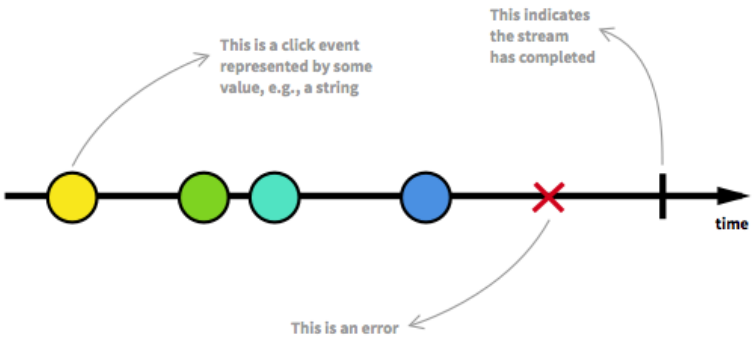
La librería RxJS nos proporciona funcionalidades para combinar, filtrar, crear... flujos. Un flujo puede ser entrada de otro, varios flujos pueden generar otro... También es importante saber que los flujos son inmutables.

Para entender todo esto vamos a ver varios ejemplos:

Contar click de un ratón.

Para empezar, vamos a crear un flujo para los click del ratón el cual puede emitir 3 cosas:

- Un valor (Será cuando se realice un click).
- Da un error.
- Señal de completados. (Esto puede suceder cuando se cierre la ventana o vista que lo contiene se cierra).



Como podemos observar en la figura de arriba, un flujo es también una secuencia de eventos ordenados en el tiempo.

Estos eventos son emitidos de forma asíncrona, para nuestro ejemplo crearemos una función para cada uno de los tres tipos de venta (valores, error y completado (Para las dos últimas en algunos casos no es necesario crear esas funciones)).

Una vez visto todo lo anterior, vamos a definir el patrón observador, para RxJS: Escuchar el flujo le vamos a llamar suscribir, las funciones que estamos definiendo son observadores y el flujo es el sujeto (o “observable”) siendo observador.

Ahora para los ejemplos vamos a usar la siguiente sintaxis en ASCII.

```
--a---b-c---d-----X---|->
```

a,b,c,d: *Son los eventos emitidos*

X: *Es un error*

|: Es la señal de que se ha completado el flujo.

--->: Es la línea de tiempo. (Cada - es una unidad de tiempo).

Ahora que ya hemos explicado cómo funciona un flujo vamos a ver el ejemplo:

Objetivo: Vamos hacer un flujo que cuenta el número de veces que hace click.

Para poder realizar este ejemplo nos vamos apoyar en las siguientes operaciones que nos proporciona la librería:

- map: Esta función genera un nuevo flujo (los flujos son inmutables), realizando una transformación por cada elemento del anterior.
- scan: Acumula valores y crea una secuencia con valores intermedios.

Para nuestro ejemplo tendremos los siguientes flujos:

- clickStream: Es el flujo que pilla los eventos de click del componente.
- contadorStream: Es el flujo donde guardaremos el contador.

Usaremos para los flujos el sufijo Stream, que es el término inglés.

Secuencia de operaciones.

```
clickStream:     ---c----c--c----c-------c--->
                 vvv map(c=>1);vvvvvvvvvvvvvvv
                 ---1----1--1----1-------1--->
                 vvvv scna(g);vvvvvvvvvvvvvvv
contadorStream:  ---1----2--3----4-------5--->
```

En el ejemplo anterior hemos visto los pasos de la secuencia y como llamando al map y al scan conseguimos crear una secuencia con los contadores. Si lo pasamos a código nos queda lo siguiente.

contadorStream = clickStream.map(c=>1).scan(g);

Ahora vamos a detallar cada operación:

- El map(c=1) lo que hace es crear por cada evento una secuencia de 1.
- El scan(g) se alimenta de las secuencias de 1 un se acumula creando la secuencia intermedia.

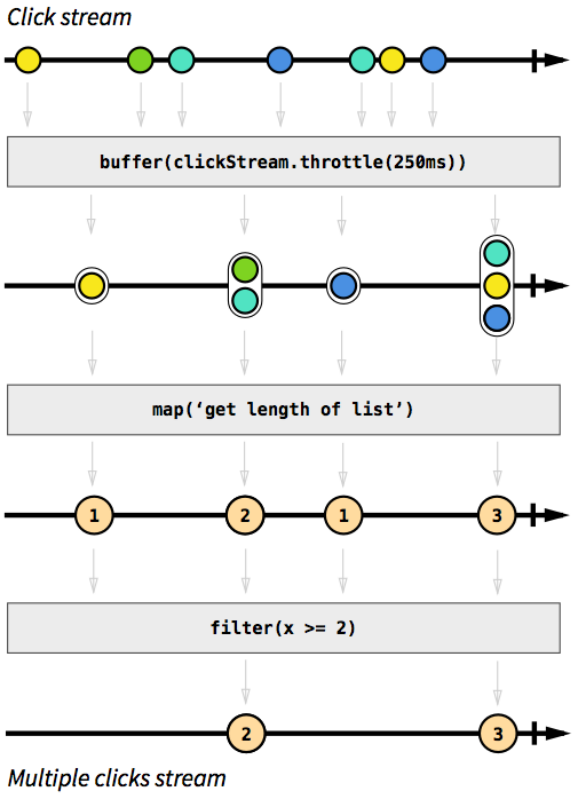
Ejemplo para detectar el doble click

Ahora vamos a ver un ejemplo algo más complejo para ver la potencia de esta librería. En este caso creamos un stream que nos indique si hemos realizado un doble click o alguno mas

La idea: es por un lado agruparlos por el tiempo (250 mls) en el tiempo los eventos en una lista, luego los transformamos la lista en el número de elementos (esto nos dará los número de elementos) y por último filtraremos los que tengan 2 o más click.

Una vez que hemos explicado de palabra el algoritmo reactivo vamos a ver el código.

clickStream.buffer((_)=> clickStream.throttle(250)).map(list=>list.length).filter(x=> x>=2=)
Esto con más detalle lo podemos ver explicado en la siguiente figura



Por último, nos suscribimos a esta secuencia para tratar el número de eventos.

¿Por qué debemos considerar la programación reactiva?

Este paradigma de programación, nos permite elevar el nivel de abstracción, para poder preocuparnos de la lógica de negocio o presentación (Ya que en el navegador el 99%, va a ser lógica de presentación).

Teniendo en cuenta que las aplicaciones han evolucionado de pantallas devueltas desde el servidor a un página con multitud de eventos en tiempo real, en este caso este paradigma se adapta muy bien.

En nuestro caso además el marco de trabajo de angular 2, en el componente http se apoyan en esta librería para saber cuándo nos llegan las peticiones del servidor.

Solicitud y respuesta

Este caso tiene que realizar las siguientes acciones:

- Hacer una petición.
- Obtener una respuesta.
- Presentación de la respuesta

Para explicar el patrón vamos a ver paso por paso como programar de forma reactiva.

Lo primero es crear un flujo con la petición, este flujo tendrá una sola respuesta.

```
--a-------|->
a: es la url (https://api.github.com/users)
```

Cada vez que se produce un evento, nos indica dos cosas:

- cuando: Es cuando se ha emitido el evento.
- Que: En este caso es la url que vamos a pedir.

En RxJS se codifica de la siguiente forma

```
var requestStream = Rx.Observable.just('https://api.github.com/users');
```

Ya hemos visto cómo realizar la solicitud, ahora nos queda ver cómo recuperar el valor, para lo cual lo primero que tenemos que hacer es suscribirnos a la emisión del evento para realizar la petición request.

```
requestStream.subscribe(function(requestUrl) {
  // execute the request
  jQuery.getJSON(requestUrl, function(responseData) {
    // ...
  });
}
```

De momento si nos fijamos aún no hemos avanzado mucho, la verdad que hemos cambiado la invocación a una función a un lanzamiento de un evento. La idea es que la petición es que nos devuelve un stream.

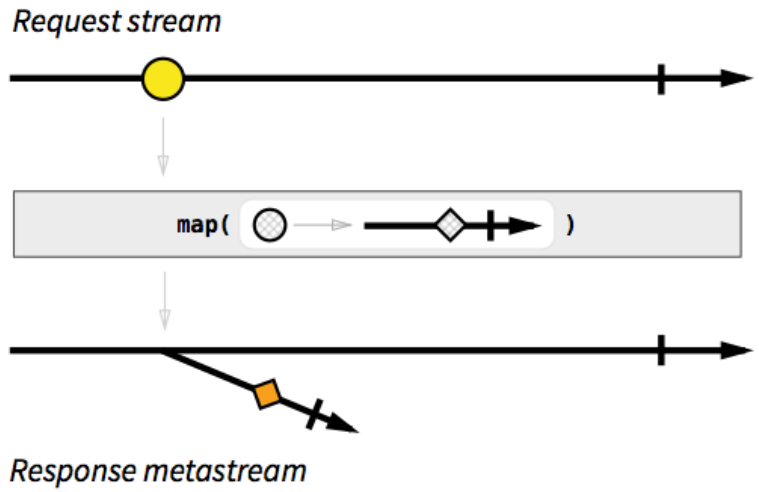
```
requestStream.subscribe(function(requestUrl) {
  // execute the request
  var responseStream = Rx.Observable.create(function (observer) {
    jQuery.getJSON(requestUrl)
    .done(function(response) { observer.onNext(response); })
    .fail(function(jqXHR, status, error) { observer.onError(error); })
    .always(function() { observer.onCompleted(); });
  });

  responseStream.subscribe(function(response) {
    // do something with the response
  });
}
```

Ahora ya somos capaces de crear un stream con la respuesta y el objeto json. Pero lo objetos observables en RxJS son promesas, por lo cual la librería nos permite hacerlo de manera automática como vamos a ver en el siguiente ejemplo.

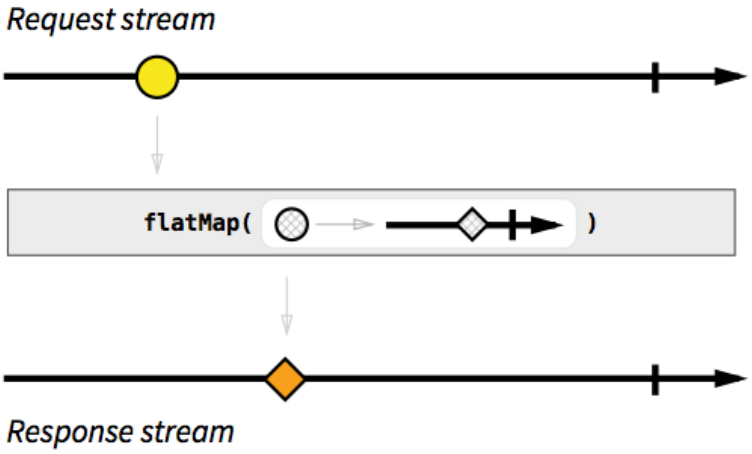
```
var responseMetastream = requestStream
  .map(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

Con la conversión de la promesa a un objeto observable, pero aún no hemos terminado de afinar la película, ya que este código en vez de crearme un flujo nos crea una bestia llamada meta Flujo que no es nada más que un flujo que por cada evento es un flujo.



Si cambiamos el map, por el flatMap, ya no nos crear un metaStream, sino que lo aplana nos devuelve un solo stream.

```
var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```



La secuencia que hemos creado:

```
requestStream:   --a-----b--c--------------------------|->
responseStream: -----A-------B------------------C-----|->
minusculas son las peticiones, mayúsculas respuestas.
```

Por cada URL se genera un evento que genera una respuesta ordenada en el tiempo

A continuación, vamos a poner la solución para esta casuística

```
var requestStream = Rx.Observable.just('https://api.github.com/users');


var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });


responseStream.subscribe(function(response) {
  // render `response` to the DOM however you wish
});
```

.

Botón de actualizar:

Cada vez que hacemos click en el botón, se debe emitir una nueva url.

OJO: Cualquier cosa puede ser un flujo.

Un flujo de eventos click, en el botón de actualizar. RxJS tiene una operación para los eventos del botón (fromEvent)

```
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');
```

Ahora necesitamos mapear a cada evento del click a una URL + más un desplazamiento, para que en cada golpe nos devuelve un conjunto distinto usuarios.

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

Necesitamos 2 comportamientos:

- Uno para cuando se abra por primera vez, lo cargue sin que tengamos el evento de click.
- Otro para el resto de click de refrescos.

Estos dos flujos los vamos a fusionar para eso vamos a usar la operación merge, que funciona como en el siguiente ejemplo:

```
flujo A:   ---a------------e------o----->
flujo B:   -----A----C--------D--------->
           vvvvvvvvv  merge vvvvvvvvvvvvv
           ---a-A----C-----e--D---o----->
```

A continuación, vemos el ejemplo en el mezclamos los dos ejemplos:

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
```

```
    return 'https://api.github.com/users?since=' + randomOffset;
  })
  .merge(Rx.Observable.just('https://api.github.com/users'));
```

Aun así podemos resolver el problema de una manera mucho más simplificada utilizando la operación ***startwitch***, lo cual nos añadirá un primer evento a la secuencia y nos resuelve la primera cargar en el formulario, sin haber hecho click.

```
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');var re-
questStream = refreshClickStream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

Ya hemos visto cómo solucionar el problema del botón actualizar, con programación reactiva y la primera carga de los datos antes del evento.

Modelar 3 sugerencias del flujo

Ya tenemos un flujo que nos genera las url, está la enganchamos al ejemplo de petición con el response.

Ahora nos toca ver como se pintar 3 elementos.

La primera idea es suscribir al flujo de url que limpie los datos, pero esto es una mala práctica ya que lo que hacemos es, distribuir en el código la responsabilidad de pintar el dom. Por lo cual vamos lo que vamos a plantear es dejarlo en un solo sitio por lo cual del response vamos a generar un flujo para pintar en este ejemplo (Crearemos 3 uno por cada línea) (Esto se puede abstraer).

Lo que vamos hacer es que ese flujo nos devuelve un json con los datos, en el JSON

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  });
```

```
suggestion1Stream.subscribe(function(suggestion) {
  // render the 1st suggestion to the DOM
});
```

Para que el refresco, limpie la sugerencia, lo que haces es mezclar con la secuencia de los click a nulos, de esta manera cuando en la subscripción si es nulo limpiamos los datos de pantalla y en caso contrario pintamos la información en el dom.

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  })
  .merge(
    refreshClickStream.map(function(){ return null; })
  );
```

```
suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
```

```
  }
});
```

Configuración del proyecto web

En este apartado vamos a ver lo que necesitamos para crear un proyecto spi.

- Herramientas y cómo configurarlo.
- Marcos de trabajo o librerías y cómo configurarlos.
- Desarrollos funcionalidades horizontales.

Herramientas:

Para el desarrollo de la aplicación vamos a utilizar las siguientes herramientas:

- nodejs.
- atom + pluging de TypeScript.

nodejs

> *Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google*

En nuestro caso lo vamos a utilizar para:

- Gestionar y descargar las librerías de javascript.
- Lanzar los test unitarios.
- Lo utiliza la herramienta atom.

Para montar el entorno vamos a seguir los siguientes pasos:

1º Instalamos la herramienta:

Nos bajamos el instalador de la página web https://nodejs.org/en/

2º Configuramos las librerías

Una de las ventajas de utilizar nodejs, es que nos proporciona npm para poder descargar las diferentes librerías y plugins que vamos a necesitar.

En la siguiente tabla vamos a ver las dependencias de la parte del navegador del proyecto:

Librería	Versión	Descripción
angular2	2.0.0-beta.0	El marco de trabajo angular 2
bootstrap	^3.3.6	Es la librería css, que vamos a utilizar.
es6-shim	^0.33.13	Nos permite compatibilidad, con JavaScript 6.
systemjs	0.19.2	Es un cargado de módulos universal (ES6 module, AMD, CommonJS)...
RxJS	5.0.0-beta.0	Librería que vamos a utilizar para la programación reactiva y además es una dependencia de angular2.

En la siguiente tabla veremos las dependencias para el desarrollo.

Librería	Versión	Descripción
typescript	^1.7.3	Es el lenguaje de TypeScript.
live-server	^0.8.2	Es un pequeño servidor web, que usaremos.

Para poder configurar todo esto vamos a establecerlo en el fichero, package.json

package.json

```
{
  "name": "angular",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "tsc": "tsc -p src -w",
    "start": "live-server --open=src"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
```

```
  "dependencies": {
    "angular2": "2.0.0-beta.0",
    "bootstrap": "^3.3.6",
    "es6-shim": "^0.33.13",
    "systemjs": "0.19.2",
    "rxjs": "5.0.0-beta.0"
  },
  "devDependencies": {
    "live-server": "^0.8.2",
    "typescript": "^1.7.3"
  }
}
```

Con este paso ya tenemos configurado todo lo que necesitamos a nivel de nodejs, ahora con ejecutar el comando npm install, se crear un directorio ./node_modules donde cargara todas las dependencias

atom

Es un editor que vamos a usar, para trabajar con los ficheros de JavaScript y ts. Lo he elegido debido a que eclipse no se lleva bien con el JavaScript y el atom tiene soporte para TypeScript y github que es nuestro control de versiones.

Al atom le vamos a instalar el paquete atom-typescript.

Marcos de trabajo o librerías

Ya los hemos explicado en el estudio teórico, pero los vamos a volver a nombrar:

- Angular 2.
- RxJS.

angular 2

Para configurar el proyecto lo primero que tenemos que saber son los módulos que vamos a usar en el proyecto que en nuestro caso son los siguientes.

- core: Es el módulo común que tiene que llevar todo proyecto.

- router: Nos cargara la funcionalidad de enrutador, que usaremos para la funcionalidad de multipantalla.
- http: Es el encargado de comunicarse con el servidor.

Una vez que sabemos las partes de angular que vamos a usar, vamos a ver la estructura mínima para un proyecto angular que se compone en los siguientes ficheros:

*** index.html
*
** boot.ts
*
** app.ts

Ahora vamos a ver cómo crear cada uno de los ficheros en el orden de importancia de más o menos el index.html invoca al boot.ts y el boot.ts invoca al app.ts.

INDEX.HTML

Este será nuestra única página, por lo cual aquí tendremos que poner las dependencias que se van a cargar en el navegador, tanto de js y como css. Por otro lado, es la encargada de empezar el proceso de carga de los datos utilizando la librería systemjs, para la carga de módulos y como último le tenemos que indicar cuál es la etiqueta del componente de aplicación.

Vamos a configurar las siguientes librerías o marcos de trabajo:

- bootstrap
- angular2
- systemjs

BOOTSTRAP:

Es la librería css que vamos a usar para maquetar nuestra aplicación:

```
<link rel="stylesheet" href="../node_modules/bootstrap/dist/css/bootstrap.min.css">
```

```
<script src="../node_modules/bootstrap/dist/js/bootstrap.js"></script>
<script src="https://ajax.goog-
leapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
```

Importamos la librería de jquery la necesitamos ya que la utiliza bootstrap.js

ANGULAR 2:

Aquí pondremos los js del core más los de los módulos que vamos a utilizar.

```
<script src="../node_modules/angular2/bundles/angular2.dev.js"></script>
<script src="../node_modules/angular2/bundles/angular2-polyfills.js"></script>
<script src="../node_modules/angular2/bundles/router.dev.js"></script>
<script src="../node_modules/angular2/bundles/http.dev.js"></script>
```

SYSTEMJS:

Estas son las librerías de js del proyecto:

```
<script src="../node_modules/systemjs/dist/system.src.js"></script>
```

RxJS

Funcionalidades horizontales

Aquí vamos a ver las diferentes desarrollos generales que hemos creado para facilitar el trabajo o generalizar ciertas operaciones.

- Componente para las ventanas modales.

Componente para las ventanas modales

Hemos creado un api para poder trabajar con las ventanas modales tendremos de 3 tipos

- Alert: Nos muestra un mensaje, al usuario.
- Confirmar: Nos muestra una ventana modal de confirmar.
- Cargar un componente: Nos pinta en la ventana modal, un componente de angular

USO

Vamos a ver cómo podemos usar la ventana modal, para lo cual vamos a ver cómo podemos llamar al módulo de las ventanas modales.

```
import
{
DbpDialogo,
DbpDialogoAlertConf,
DbpDialogoConfirmarConf,
DbpDialogoBaseConf,
DbpDialogoRef
}
from '../../../core/modal/dialogo';
```

Hemos creado 3 tipos de ventanas modales:

- **Alert**: Nos muestra un mensaje, al usuario.
- **Confirmar**: Nos muestra una ventana modal de confirmar.
- **Componente**: Nos pinta en la ventana modal, un componente de angular.

Estas operaciones, se encuentran en la clase (***DbpDialogo***), será nuestra api.

ALERT:

Esta ventana modal, nos mostrará un mensaje.

Para configurar, usaremos el objeto (DbpDialogoAlertConf), que contiene los siguientes atributos:

- **título:** Es el título que vamos a poner en la ventana modal.
- **mensaje:** Es el mensaje que vamos a mostrar.

Para crear una ventana modal de tipo Alert, llamaremos a la siguiente función en la clase ***DbpDialogo***

```
public alert(elemento:ElementRef,dialogoConf:DbpDialogoAlertConf)
    :Promise<DbpDialogoRef>
```

Parámetros

- ***elemento: ElementRef***: Le pasamos el componente que va invocar la ventana modal. (Se puede inyectar en el constructor del componente).
- ***dialogoConf: DbpDialogoAlertConf***: Le pasamos la configuración, para los alert.

A continuación vamos a ver ejemplo de uso, en el cual se pintara una ventana modal, con el título '*ejemplo alert*' y el texto '*Esto es un ejemplo de ventana modal*'.

```
import
{
```

```
DbpDialogo,
DbpDialogoAlertConf,
DbpDialogoRef
}
from '../../../core/modal/dialogo';
...
constructor(private elemento:ElementRef,private dialogo:DbpDialogo){
}

  abrirAlert(){
    this.dialogo.alert(this.elemento,
        new DbpDialogoAlertConf(
                'Esto es un ejemplo de ventana modal',
                'ejemplo alert'))
        .then(dialogoRef=>{
            dialogoRef.cuandoCerramos
              .then((_)=>{
                  console.info('Se ha cerrado el alert')
              });
      return dialogoRef;
    });
  }
```

El método nos devuelve una promesa que se ejecutará cuando se ha creado, además en este ejemplo nos vamos a enganchar, a la promesa de cierra, para que se ejecute un código cuando se cierra.

CONFIRMAR

Nos mostrará un mensaje, y nos permite realizar la operación de confirmar o cancelar.

Para configurar, usaremos el objeto (***DbpDialogoConfirmarConf***), que contiene los siguientes atributos:

- **título:** Es el título que vamos a poner en la ventana modal.
- **mensaje:** Es el mensaje que vamos a mostrar.
- **botonOk:** El texto que pondremos en el botón ok, por defecto pondrá el texto 'Ok'
- **botonCancelar:** El texto que pondremos en el botón de cancelar, por defecto pondrá el texto 'Cancelar'

Para crear una ventana modal de tipo confirmar, llamaremos a la siguiente función en la clase *DbpDialogo*

```
 public confirmar(elemento:ElementRef,dialogoConf:DbpDialogoConfirmarConf)
:Promise<ComponentRef>{
```

Parámetros

- ***elemento: ElementRef***: Le pasamos el componente que va invocar la ventana modal. (Se puede inyectar en el constructor del componente).
- ***dialogoConf: DbpDialogoConfirmarConf***: Le pasamos la configuración, para la confirmación.

A continuación vamos a ver un ejemplo de uso, en la cual se pintara una ventana modal, para confirmar una operación, y tendremos dos promesas asociadas a la operación, para saber cuándo le damos a ok o a cancelar:

```
import
{
DbpDialogo,
DbpDialogoConfirmarConf,
DbpDialogoRef
}
from '../../../core/modal/dialogo';
...
constructor(private elemento:ElementRef,private dialogo:DbpDialogo){
}

  abrirConfirmar(){
        this.dialogo.confirmar(this.elemento
        ,new DbpDialogoConfirmarConf(
                    'Mensaje de confirmar'
                   ,'Ejemplo de confirmar'))
             .then(dialogoComponent=>{
                   dialogoComponent.instance.cuandoOk
                      .then((_)=>{
                        console.info(' despues Ok 234');
                       });
                   dialogoComponent.instance.cuandoCancelar
                      .then((_)=>{
                        console.info(' despues cancelar 234');
                       });
    });
```

```
  }
```

La promesa ***cuandoOk***, se ejecutara su clausura, cuando pulsemos el botón ok. La promesa ***cuandoCancelar***, se ejecutara su clausura cuando pulsemos el botón cancelar.

COMPONENTE

En este ventana modal podemos inyectar un componente de angular 2. (Admite la posibilidad, de poder inyectarle cosas al contexto del componente y nos podemos recuperar el componente al cerrar la ventana). Este tipo de ventana modal nos permite personalizar como nos dé la gana.

Para configurar, usaremos el objeto (***DbpDialogoBaseConf***).

- **título:** Es el título que vamos a poner en la ventana modal.

Para crear una ventana modal de componente, llamaremos a la siguiente función en la clase ***DbpDialogo***

```
 public abrir(tipo:Type,elemento:ElementRef,dialogoConf:DbpDialogoBaseConf
,providers: Array<Type | Provider | any[]> = [])
:Promise<DbpDialogoRef>{
}
```

Parámetros

- ***tipo: Type***: Es la clase que representa el componente que vamos a inyectar.
- ***elemento: ElementRef***: Le pasamos el componente que va invocar la ventana modal. (Se puede inyectar en el constructor del componente).
- ***dialogoConf: DbpDialogoBaseConf***: Le pasamos la configuración, para la confirmación.
- ***providers: Array = []***: Aquí le pasaremos los objetos que vamos a inyectar al sistema.

Ojo: El componente se puede inyectar la referencia a la ventana modal con la que estamos trabajando ***DbpDialogoRef***

A continuación vamos a ver diferentes ejemplos o casuísticas

- Crear un componente simple.
- Crear un componente, al que le pasamos un dato de entrada y recuperar una salida.

COMPONENTE SIMPLE

Lo primero que vamos a hacer es crear el componente en cuestión

```
@Component({
  selector:'ejemplo-simple',
  template:'Ejemplo de componente simple'
})
export class EjemploSimpleComponent{
}
```

Ahora vemos cómo vamos a invocar la ventana modal.

```
import
{
DbpDialogo,
DbpDialogoBaseConf,
DbpDialogoRef
}
from '../../../core/modal/dialogo';
...
constructor(private elemento:ElementRef,private dialogo:DbpDialogo){
}

  abrirComponenteSimple(){
     this.dialogo.abrir(EjemploSimpleComponent,this.elemento,
          new DbpDialogoBaseConf('Ejemplo para'))
          .then(dialogoRef=>{
            return dialogoRef;
           });
  }
```

COMPONENTE ENTRADA/SALIDA

Lo primero es crear el componente

```
@Component({
  selector:'ejemplo-form',
  template:`
     Ejemplo de componente complejo
<input [(ngModel)]="dato">
<button type="button" class="btn btn-primary" (click)="ok()">
  Ok
```

```
</button>
`
})
export class EjemploFormComponent{
  public dato:String;
  constructor(id:ParamId,private dbpDialogoRef:DbpDialogoRef){
     this.dato=id.id;
  }

  ok(){
    this.dbpDialogoRef.cerrar();
  }

}

export class ParamId{
  constructor(public id:Number){}
}
```

Ahora vemos cómo vamos a invocar la ventana modal.

```
import
{
DbpDialogo,
DbpDialogoBaseConf,
DbpDialogoRef
}
from '../../../core/modal/dialogo';
...
constructor(private elemento:ElementRef,private dialogo:DbpDialogo){
}

abrirComponenteComplejo(){
     var id=[provide(ParamId, {useValue:new ParamId(2)})];
    this.dialogo.abrir(EjemploFormComponent,this.elemento
      ,new DbpDialogoBaseConf('Ejemplo para'),id)
      .then(dialogoRef=>{
           console.info('Componente de dentro',dialogoRef.componenteDentro);
           dialogoRef.cuandoCerramos.then((_)=>{
                  console.info('Se cerro el componente'
                   ,dialogoRef.componenteDentro.instance.dato);
```

```
            });
        return dialogoRef;
    });
}
```

Le pasamos al controlador el parámetro Id, para lo cual usaremos un provide, y cuando cerramos si accedemos dialogoRef.componenteDentro.instance (es el componente que hemos abierto dentro de la ventana modal). Y si accedemos al atributo dato (Como la salida del formulario).

DESARROLLO

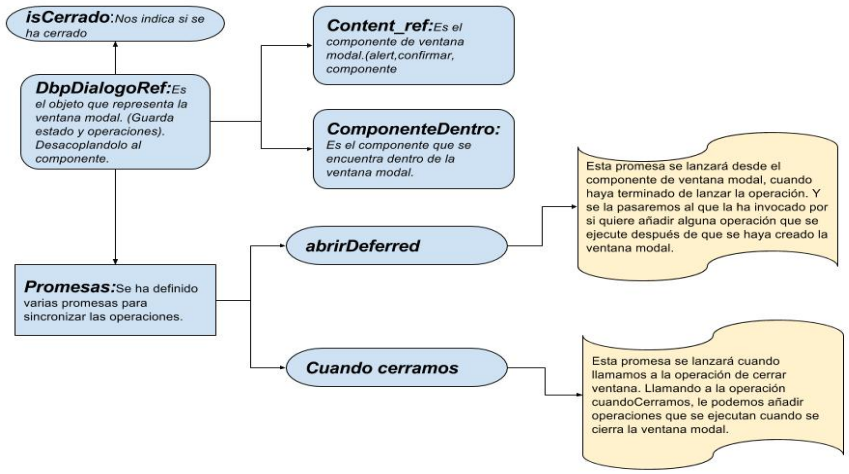
Para poder trabajar en angular 2, hemos creado un generalidad horizontal, para poder trabajar con las ventanas modales. Por cada tipo de ventana modal hemos creado:

- ***Un componente***: Que es el que representará la ventana modal.
- ***Una plantilla***: Se ha creado una plantilla por cada ventana modal dialogo.html
- ***Un fichero de configuración***: Por cada tipo tenemos un fichero de configuración.

Para poder canalizar esto tenemos los diferentes tipos de clases:

- ***DbpDialogo***: Esta es la clase que funcionara como un api, con la que vamos a interactuar, desde fuera.
- ***DbpDialogoRef***: Esta es la clase que va a mantener el estado, las operaciones y los componentes relacionados con la ventana modal. (Esta abstracción nos permite centralizar en un solo objeto común, los diferentes tipos de componentes y el estado que solo tenga que ver con la ventana modal). También tendremos dos promesas, una de ellas se lanzará cuando se ha cargado la ventana modal, y la otra se lanzará cuando se ha cerrado la ventana modal. A estas dos promesas podemos siempre lanzar operaciones. Este objeto siempre lo podremos inyectar en el componente de dentro de la ventana modal y de esa manera sabremos si está en una ventana modal o no.

En el siguiente diagrama podemos ver cómo está compuesto el objeto **DbpDialogoRef**



Lo importante es explicar cómo funciona el ciclo de vida de una ventana modal:
Vamos a ver los pasos que seguimos para crear una ventana modal:

1. Ejecutamos el componente encargado, de bloquear toda la pantalla (***BlockDialogo-Component***).
2. Creamos el componente de ventana modal, que corresponda según el tipo de ventana.
3. Le pasamos el fichero de configuración, y el fichero de referencia. (Esto queda pendiente de pasárselo como contexto.
4. Vinculamos el cierre de la ventana modal al de quitar el bloqueo de pantalla.
5. Vinculamos el componente, y lanzamos las promesas que se ha creado la ventana modal.

Spring

Es el marco de trabajo que nos va a gestionar todo, en el servidor.

¿qué es spring?

Spring es un marco de trabajo de código abierto creado por Rob Johnson. Fue creado para tratar la complejidad del desarrollo de aplicaciones empresariales. Spring hace posible utilizar JavaBean sencillos para conseguir cosas que antes sólo eran posibles con EJB. No obstante, la utilidad de spring no está limitada al desarrollo en el extremo del servidor. Cualquier aplicación Java puede beneficiarse de Spring en términos de simplicidad y acoplamiento débil.

Spring hace muchas cosas, pero cuando se desmonta en sus partes básicas, Spring es un contenedor y marco de trabajo ligero de inyección de dependencia y orientado a aspectos. es mucho, pero resume bien el propósito principal de Spring. Para tener un mejor visión de lo que es Spring, desglosemos esta descripción.

- **Contenedor**: Spring es un contenedor en el sentido de que contiene y gestiona el ciclo de vida y configuración de objetos de aplicación. En Spring, puede declarar cómo debe crearse cada objeto de su aplicación, cómo deben configurarse y cómo deben asociarse entre sí.
- **Marco de trabajo**: Spring hace posible configurar y escribir aplicaciones complejas desde componentes sencillos. En Spring, los objetos de aplicación se escriben de forma declarativa, normalmente en un archivo XML (actualmente lo podemos hacer todo con anotaciones, es una decisión del equipo de desarrollo que una imposición, del marco de trabajo). Spring también proporciona gran funcionalidad de infraestructura (gestión de transacciones, integración con el marco de trabajo de persistencia, etc.) dejándole a usted el desarrollo de la lógica de aplicación.
- **Ligero**: Spring es ligero tanto en términos de tamaño como en tiempo de procesamiento. El volumen del marco de trabajo Spring puede distribuirse en un único archivo JAR que pesa poco más de 2,5MB. Y el tiempo de procesamiento requerido por Spring es prácticamente insignificante. Es más, Spring no es intrusivo: los objetos de una aplicación habilitada para Spring a menudo no tienen dependencias de clases específicas de Spring.
- **Inyección de dependencia:** Spring fomenta el acoplamiento débil mediante una técnica conocida como inyección de dependencia (DI). Cuando se aplica la DI, se otorga a los objetos de forma pasiva sus dependencias, en lugar de crear o buscar

objetos dependientes por sí mismos. Puede pensar en DI como JNDI al contrario; en lugar de que un objeto busque dependencias de un contenedor, el contenedor otorga las dependencias al objeto en la creación de la instancia sin esperar a que se pida.

- **Orientado a aspectos**: Spring tiene un amplio soporte para programación orientada a aspectos (AOP) que permite el desarrollo cohesivo separando la lógica empresarial de la aplicación de los servicios de sistemas (como auditoría y gestión de transacciones). Los objetos de aplicación hacen lo que se suponen que deben hacer (realizar la lógica empresarial) y nada más. No son responsables (o ni siquiera conscientes) de otros aspectos del sistema, como los registros o soporte transaccional.

Dicho de otra forma: cuando se desglosa Spring en sus partes básicas, lo que se obtiene es un marco de trabajo que ayuda a desarrollar código de aplicación de acoplamiento débil. Incluso si eso fuera todo lo que hace Spring, las ventajas de acoplamiento débil (capacidad de mantenimiento y comprobación) harían de Spring un marco de trabajo valioso sobre el que construir aplicaciones.

Pero Spring, es más. El marco de trabajo Spring tiene varios módulos basados en DI y AOP para crear una plataforma llena de atributos sobre lo que construir aplicaciones.

La vida de un bean

Los pasos que se van a seguir para crear un bean son los siguientes:

1. Spring instancia el bean.
2. Spring inyecta valores y referencia el bean en las propiedades de éste.
3. Si el bean implementa BeanNameAware, Spring proporciona el ID del bean al método setBeanName().
4. Si el bean implementa BeanFactoryAware, Spring ejecuta el método setBeanFactory. proporcionando el mismo la fábrica de bean.
5. Si el bean implementa AplicationContextAware, Spring ejecutará el método setApplicationContext(), proporcionándolo en una referencia al contexto de aplicación contenedor.
6. Si cualquiera bean implementa la interfaz BeanPostProcessor, Spring ejecuta su método postProcessBeforerInitialization().
7. Si cualquier bean implementa la interfaz InitializingBean, Spring ejecuta su método afterPropertiesSet() método. De forma similar, si el bean se ha declarado con un init-method, entonces se ejecuta el método de inicialización especificado.

8. Si hay algún bean que implementa BeanPostProcessor, Spring va a ejecutar su método postProcesssAfterInitializaction().
9. Llegados a este punto, el bean está listo para que la aplicación lo utilice, y va a permanecer en el contexto de la aplicación hasta que se elimine.
10. Si algún bean implementa la interfaz DisposableBean, Spring ejecutará sus métodos destroy(). De la misma manera, si se ha declarado algún bean con destroy-method, entonces se ejecutará el método especificado.

Arquitectura de la parte servidora

En nuestro proyecto, la parte de back-end, la vamos a dividir en 3 capas:

- Dao: Es la capa del acceso a datos.
- Service: Es la capa donde tendremos la lógica de negocio.
- Vista: En nuestro caso, serán servicios REST.

A continuación, vamos a ver las configuraciones y generalidades que vamos a necesitar para las diferentes capas.

Configuración de la capa de persistencia

Objetivo

Un requisito que casi todas las aplicaciones empresariales tienen: la persistencia de los datos. En la práctica, el acceso a los datos presenta muchas dificultades. Hay que inicializar nuestro marco de trabajo de acceso a los datos, abrir conexiones, gestionar varias excepciones y cerrar conexiones. Si algunos de estos pasos no se hacen de forma correcta, podríamos corromper o eliminar valiosos datos de nuestra empresa.

Para gestionar este problema, spring cuenta con una familia de marcos de trabajo de acceso que se integran con una variedad de tecnologías de acceso de acceso a los datos. La ventaja de usar Spring, es que con independencia de que tecnología utilicemos para persistir los datos mediante JDBC directo, IBATIS o un ORM como hibernate, spring nos libera de parte del trabajo pesado del acceso a los datos dentro del código de persistencia. Vamos a dejar a spring el acceso a los datos de bajo nivel y nos vamos a centrar su atención en la gestión de los datos de la aplicación.

Decisiones

En la capa de persistencia tenemos que tomar la decisión de donde vamos a persistir los datos, si en una B.D relacional clásica o en una Base de datos clave valor, en nuestro caso

al ser una aplicación empresarial clásica nos interesa una B.D de datos relacional, ya que lo más importante es la consistencia de los datos.

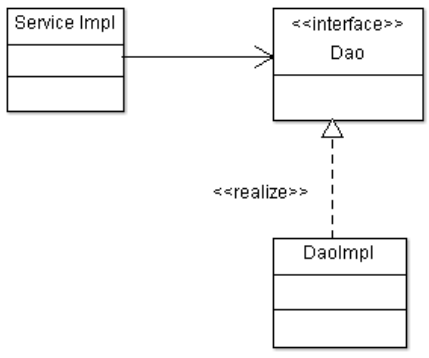
Una vez que hemos elegido que vamos a trabajar con una B.D relacionar ahora vamos a ver que B.D vamos a utilizar, como en nuestro caso vamos a usar un ORM, para la capa de persistencia, como para este proyecto es una ver un ejemplo de cómo se usa la tecnología vamos a optar, por una B.D embebida ***HSQLDB.***

Ahora que ya sabemos que vamos a usar una B.D de datos relacionar vamos a utilizar un ORM, y dentro de los diferentes ORM que hay en el mercado hemos optado por hibernate ya que es una solución consolidada en el mercado, y que sigue actualizándose todos los años y adaptándose a las nuevas especificaciones de JPA.

Filosofía de acceso a datos de spring.

Teniendo en cuenta que uno de los principales objetivos de spring es permitir el desarrollo de aplicaciones siguiendo el principio de programación de interfaces Orientadas a objetos (OO). El acceso a datos de spring no es una excepción

Para lo cual en spring vamos a utilizar DAO (Objeto de acceso a datos), los cuales existen para proporcionar un método para leer y escribir datos en la Base de Datos. Estos deben exponer esta funcionalidad mediante una interfaz, a través de la cual el resto de la aplicación va a tener acceso a ellos.



Configuración

Ahora que sabemos, las herramientas que disponemos, y a dónde queremos ir. Una de las características que tiene spring es que nos permite dividir la configuración en diferentes ficheros, para facilitar la configuración del framework, vamos a crear el fichero de configuración (***JpaConfig***).

En el fichero vamos a configurar el datasource, que es la forma que tenemos de indicarle al sistema donde se encuentra nuestra B.D.

```
	@Bean
	public DataSource getDataSource() {
	    final BasicDataSource dataSource = new BasicDataSource();
	      dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
	      dataSource.setUrl("jdbc:hsqldb:mem:pruebas");
	      dataSource.setUsername("sa");
	      dataSource.setPassword("");
	    return dataSource;
	}
```

Ahora que ya el sistema sabe cómo acceder a la B.D, vamos a configurar nuestro ORM, aquí simplemente le tenemos que indicar por un lado en que paquete van a estar las entidades de la Base de datos. (org.dbp.bom). Y por otro lado le pasaremos diferentes configuraciones del ORM:

- Le vamos a indicar que cada vez que se cree la aplicación creara la B.D desde cero.
- Le indicamos que nos muestre las instrucciones SQL que se generan (No es una buena opción para producción).
- Le tenemos que indicar el dialecto con el que vamos a trabajar, para entenderse con la B.D., (org.hibernate.dialect.HSQLDialect).
- Por ultimo le indicamos los scripts de SQL que queremos que se carguen al crear la B.D.

```
	/**
	 * Configuramos el entity manager en JPA. - Le indicamos el data
source con
	 * el que va a trabajar. - Le indicamos el paquete donde se encuen-
tran las
	 * clases. - Le pasamos el adaptador de JPA que en nuestro caso
sera
	 * hibernate. - Por otro lado le pasamos las propiedades.
	 *
	 * @return
	 */
	@Bean
	public LocalContainerEntityManagerFactoryBean entityManagerFacto-
ryBean() {
```

```
            final LocalContainerEntityManagerFactoryBean localContainer-
EntityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();
            localContainerEntityManagerFactoryBean.setDataSource(getData-
Source());
            localContainerEntityManagerFactoryBean.setPackagesToScan(new
String[] { "org.dbp.bom" });
            final JpaVendorAdapter vendorAdapter = new HibernateJpaVendo-
rAdapter();
            localContainerEntityManagerFactoryBean.setJpaVendorAdap-
ter(vendorAdapter);
            localContainerEntityManagerFactoryBean.setJpaProperties(pro-
piedadesAdicionalesJpa());
            return localContainerEntityManagerFactoryBean;
      }

      /**
       * Le indicamos el tipo de configuraciï¿½n que nos interesa para
hibernate -
       * Le indicmaos que cada vez que entremos borre y cree las B.D. -
Le
       * indicamos que utilice el dialecto con HSQL.
       *
       * Nota: este mï¿½todo no es parte de la configuraciï¿½n de spring.
       *
       * @return
       */
      @SuppressWarnings("serial")
      private Properties propiedadesAdicionalesJpa() {
            return new Properties() {
                  {
                        setProperty("hibernate.hbm2ddl.auto", "create");
                        setProperty("hibernate.jdbc.batch_size", "20");
                        setProperty("hibernate.show_sql", "true");
                        setProperty("hibernate.dialect","org.hiber-
nate.dialect.HSQLDialect");


                        setProperty("hibernate.hbm2ddl.im-
port_files","classpath:/META-INF/inicializar.sql,/META-
INF/paises.sql,/META-INF/comunidades.sql,/META-INF/provincias.sql,/META-
INF/municipios.sql");
                  }
            };
      }

      /**
```

Ahora tenemos que configurar el sistema transaccional.

```
      /**
       * Configura las transaciones en JPA.
       *
       * @return
       */
      @Bean
      public PlatformTransactionManager transactionManager(final Enti-
tyManagerFactory emf){
            final JpaTransactionManager transactionManager = new
JpaTransactionManager();
```

```
            transactionManager.setEntityManagerFactory(emf);
            return transactionManager;
        }
```

Dao genérico

Aquí vamos, a montar un dao genérico, que contenga las siguientes operaciones básicas:

- **ObtenerId**: Nos obtiene la entidad asociada a un id.
- **Eliminar**: Se encarga de eliminar la entidad que le pasamos.
- **Actualizar**: Se encarga de actualizar la entidad que le pasamos.
- **Crear**: Se encarga de crear una entidad.
- **Obtener todos**: Nos devuelve todos los registros del sistema.

La idea de un dao genérico, es que estas operaciones las hereden todos los dao de nuestro proyecto, sin necesidad de tener que picarlas a mano.
El dao genérico, se compone de una interfaz y una clase que implementa las operaciones.

De esta manera nos ahorraremos un montón de trabajo por un lado y por otro lado si necesitamos modificar algún cambio en las operaciones básicas, solo hay que hacerlo en un solo sitio.

INTERFAZ

Lo primer que tenemos que hacer es definir las operaciones, en una interfaz, en la cual además de las operaciones, vamos a definir en la cabecera de la interfaz, dos tipos genéricos, que van a representar a la entidad con la que vamos a trabajar y el id de la entidad.

Gracias a estos dos genéricos, es lo que va hacer la magia que nos va a permitir que estas operaciones valgan para cualquier de nuestras entidades del proyecto.

```
package org.dbp.core.dao;

import java.io.Serializable;
import java.util.List;

public interface GenericDao <E extends Serializable,ID>{

        E obtenerId(ID identificador);

        void eliminar(E entidad);

        void crear(E entidad);
```

```
        E actualizar(E entidad);
        List<E> obtenerTodos();
}
```

CLASE

Ahora vamos a implementar las operaciones que hemos puesto en el interfaz, en la siguiente clase.

```
package org.dbp.core.dao.impl;


import java.io.Serializable;
import java.util.List;


import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;


import org.dbp.core.dao.GenericDao;
import org.springframework.transaction.annotation.Transactional;


@Transactional(rollbackFor=Exception.class)
public class GenericDaoImpl <E extends Serializable,ID> implements GenericDao<E,ID>{


        @PersistenceContext private EntityManager entityManager;
```

```
	private final Class<E> clazzE;

	public GenericDaoImpl(final Class<E> clazzE) {
		super();
		this.clazzE = clazzE;
	}

	public E obtenerId(final ID identificador){
		return entityManager.find(clazzE, identificador);
	}

	public void eliminar(final E entidad){
		entityManager.remove(entidad);
	}

	public void crear(final E entidad){
		entityManager.persist(entidad);
	}

	public E actualizar(final E actualizar){
		return entityManager.merge(actualizar);
	}
	public List<E> obtenerTodos(){
		final CriteriaBuilder criteriaBuilder=entityManager.getCriteriaBuilder();
```

```
                final CriteriaQuery<E> criteria=criteriaBuilder.createQuery(clazzE);

                final Root<E> from=criteria.from(clazzE);

                final TypedQuery<E> query=entityManager.createQuery(criteria.se-
lect(from));

                return query.getResultList();

        }

}
```

EJEMPLO DE USO

Una vez que está definido el dao genérico, para ver toda su potencia es ver el uso y poder apreciar el ahorro. Vamos a el caso de la cuenta contable.

```
package org.dbp.dao;

import org.dbp.bom.contabilidad.CuentaContable;
import org.dbp.core.dao.GenericDao;

public interface CuentaContableDao extends GenericDao<CuentaContable, String>{

}
```

```
package org.dbp.dao.impl;

import org.dbp.bom.contabilidad.CuentaContable;
import org.dbp.core.dao.impl.GenericDaoImpl;
import org.dbp.dao.CuentaContableDao;
import org.springframework.stereotype.Repository;

@Repository
public class CuentaContableDaoImpl extends GenericDaoImpl<CuentaContable,String>
implements CuentaContableDao {

        public CuentaContableDaoImpl() {
                super(CuentaContable.class);
        }

}
```

Services

Ahora que ya tenemos resuelto el tema de la persistencia, vamos a ver dónde vamos a crear el código de la lógica de negocio, en nuestro caso simplemente utilizaremos los servicios, que además también tendremos un interfaz y serviceImpl para trasladar la lógica del negocio.

Mejoras u otras opciones

Una mejora interesante en la capa de persistencia, es utilizar el módulo de spring-data, que reduce nuestro código a crear repositorios.

Configuración del módulo de seguridad.

Objetivo:

Al ser datos privados de una empresa, no nos interesa que cualquiera pueda acceder a la información sin nuestro consentimiento. Aunque es posible crear código nativo relacionado con la seguridad, es mejor si es posible encontrarse los aspectos de la seguridad separados de los de la aplicación

Para desarrollar esta parte de la aplicación, vamos a utilizar Spring security.

Herramientas (Spring security);

Spring security, es un marco de trabajo de seguridad que proporciona seguridad declarativa para sus aplicaciones basadas en Spring. Asimismo, cuenta con una solución de seguridad integral que gestiona la autenticación y la autorización, tanto a nivel de solicitud Web como a nivel de ejecución de método. Basándose en el marco de trabajo de Spring, Spring security saca el máximo partido a la Inyección de dependencias y las técnicas de orientación a aspectos.

Spring Security hace frente a la seguridad desde dos ángulos. Para proteger solicitudes Web y restringir al acceso al nivel de URL, utiliza servlet filters. Spring security también puede proteger las invocaciones de método utilizando la AOP de Spring, aplicando un proxy sobre objetos y aplicando consejos que garanticen que el usuario cuenta con la autoridad adecuada para invocar métodos asegurados.

Definir la política de seguridad.

La aplicación va a ser privada 100%, por lo cual tendremos que configurar una página de login y dejaremos libres los recursos estáticos.

Por comodidad los usuarios, los vamos a crear en memoria.

Configurar Spring security

Lo primero vamos a configurar el authenticationManager, donde le indicaremos los usuarios válidos para nuestra aplicación

```
	@Autowired
	public void configureGlobal(final AuthenticationManagerBuilder
auth) throws Exception { //NOPMD
		auth
			.inMemoryAuthentication()
				.withUser("user").password("pass-
word").roles("USER","ADMINISTRACION","CONTABLE")
				.and()
				.withUser("contable").password("conta-
ble").roles("ADMINISTRACION")
				.and()
				.withUser("administrador").password("administra-
dor").roles("CONTABLE");
	}
```

Lo siguiente es configurar la política de seguridad que vamos a usar:

- Desactivamos el csrf (para el tema de las peticiones AJAX).
- Le indicamos que todas las peticiones de request, requieren autenticación.
- Le indicamos donde se encuentra la página de login.

```
@Override
	protected void configure(final HttpSecurity http) throws Exception
{ //NOPMD
		super.configure(http);
		http.csrf().disable() // Para las peticiones aya
			.authorizeRequests()	// Le indicamos que la autori-
zacion va a ser a nivel de request.
			.anyRequest().authenticated()  // Le indicamos que cada
solicitud requiere que el usuario de autentique.
			.and()
		.formLogin()	// Configurar el formato de login.
			.loginPage("/login") // aqui le indicamos donde se en-
cuentra la pagina de login.
			.permitAll();// Aqui le indicamos que la pagina de lo-
gin es publica.*/
	}
```

Por ultimo vamos a indicarle los recursos, que no va a estar securizados, como js y HTML estáticos…

```
    @Override
	public void configure(final WebSecurity web) throws Exception {
//NOPMD
	web
		.ignoring()
		.antMatchers("/resources/**")
		.antMatchers("/node_modules/**");
```

```
            super.configure(web);
        }
```

Nota: Esta configuración se encuentra toda en el fichero SeguridadConfig (Proyecto webapp).

Mejoras:

Aquí la mejora, seria de la pasar los usuarios a un directorio en la B.D y crear un interfaz para gestionar los usuarios desde la aplicación.

Configuración de un rest.

Lo más importante de un negocio son los datos. En los últimos años, la transferencia de estados representacionales (también conocida por sus siglas en inglés REST) se ha convertido en una alternativa muy popular centrada en la información y opuesta a los servicios Web basados en SOAP.

En el caso de spring la compatibilidad con REST se basa en Spring MVC, por lo cual para gestionar las solicitudes de recursos REST, lo haremos con un controlador de MVC, por lo cual va a ser transparente para nosotros.

Acerca de REST

En los últimos años se ha hablado mucho sobre REST. De hecho, se ha puesto de moda en los círculos de desarrollo de software hablar mal sobre los servicios Web basados en SOAP y promover REST como una alternativa.

Sin duda, SOAP puede resultar demasiado pesado para muchas aplicaciones y REST ofrece una alternativa más sencilla. Sin embargo, no todo el mundo conoce a donde este el modelo. En consecuencia, tenemos muchos usuarios desinformados. Antes de habla sobre la compatibilidad de Spring con este modelo, tenemos que entender realmente en que consiste REST.

Aspectos básicos de REST

Un error habitual a la hora de hablar sobre REST es considerarlo como “servicios Web con URL”. Es decir, como otro mecanismo para Llamadas de procedimiento remotos (RPC) como SOAP, aunque mediante el uso de URL HTTP y omitiendo los espacios de nombre XML.

En realidad, REST tiene muy poco en común RPC. Mientras que éste está orientado a servicios y se centra en las acciones y los verbos, REST se orienta hacia los recursos, enfatizando los elementos que forman parte de este concepto: “Transferencia de estado representacional”

- Transferencia: REST implica la transferencia de datos de recursos, en forma de representación, de una aplicación a otra.
- Estado: cuando trabajamos con REST, nos preocupa más el estado de un recurso que las acciones que podemos realizar con estos.
- Representacional: los recursos de REST puede representar en prácticamente cualquier formato, incluyendo XML, JSON (JavaScript Object Natation o incluso HTML (el formato que mejor se adapte al usuario de esos recursos).

En resumen, podemos decir que REST consiste en transferir el estado de los recursos, en el formato que consideramos más adecuado, de un servidor a un cliente y viceversa. Teniendo en cuenta esta visión de REST, prefiero evitar términos como “servicio REST”, “servicio Web REST” o cualquier denominación similar que, de forma incorrecta, dé más importancia a las acciones. En su lugar, prefiero destacar la naturaleza orientada a los recursos de REST y hablar de “recursos REST”.

Controlador genérico

Ya tenemos definido un dao genérico, un servicio genérico y ahora estas operaciones las vamos a publica como un rest genérico.

```
package org.dbp.core.controller;

import java.io.Serializable;
import java.lang.reflect.InvocationTargetException;
import java.util.List;
import java.util.Optional;

import javax.persistence.PersistenceException;

import org.dbp.core.service.GenericService;
```

```
import org.dbp.utils.ExceptionUtils;
import org.hibernate.TransientPropertyValueException;
import org.hibernate.exception.ConstraintViolationException;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.dao.InvalidDataAccessApiUsageException;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

public class GenericRestController <E extends Serializable,ID>{



	private GenericService<E,ID> service;
	private ObtenerId<ID,E> obtenerId;

	public GenericRestController(
			final GenericService<E,ID> service
			,final ObtenerId<ID,E> obtenerId){

		this.service=service;
		this.obtenerId=obtenerId;
	}

	public interface ObtenerId<ID,E>{
		public ID obtenerId(E entidad);
	}

	@RequestMapping(method=RequestMethod.POST)
	public ResponseEntity<E> crear(@RequestBody final E entidad) throws Instantia-
tionException, IllegalAccessException, InvocationTargetException, NoSuchMethodExcep-
tion{
		service.crear(entidad);
		HttpHeaders httpHeaders = new HttpHeaders();
		httpHeaders.setLocation(ServletUriComponentsBuilder
				.fromCurrentRequest().path("/{id}")
				.buildAndExpand(this.obtenerId).toUri());
		//throw new DuplicadoException();
		return new ResponseEntity<E>(entidad,httpHeaders, HttpStatus.CRE-
ATED);
	}
	@RequestMapping(method=RequestMethod.GET)
	public List<E> obtenerTodos(){
		return service.obtenerTodos();
	}

	@Transactional
```

```
	@RequestMapping(value="/{identificador}",method=RequestMethod.GET)
	public E obtenerId(@PathVariable final ID identificador){
			return service.obtenerId(identificador);
	}

	@RequestMapping(method=RequestMethod.PUT)
	public E actualizar (@RequestBody E entidad){
		return service.actualizar(entidad);
	}

	@RequestMapping(value="/lista",method=RequestMethod.PUT)
	public List<E> actualizar (@RequestBody List<E> entidad){
		return service.actualizar(entidad);
	}


	@Transactional(rollbackFor=Exception.class)
	@RequestMapping(value="/{identificador}",method=RequestMethod.DELETE)
	public void eliminar(@PathVariable final ID identificador){
		service.eliminar(service.obtenerId(identificador));
	}

	// Procesamiento de los errores.
	@ResponseStatus(HttpStatus.CONFLICT)  // 409
   @ExceptionHandler(DataIntegrityViolationException.class)
   public String handleConflict(DataIntegrityViolationException e) {
		Optional<ConstraintViolationException> constraint =ExceptionUtils.get-
Instancia().buscarCause(e,ConstraintViolationException.class);
		if(constraint.isPresent()){
			// códigos de error de HSQLDB ...
http://grepcode.com/file/repo1.maven.org/ma-
ven2/org.hsqldb/hsqldb/2.3.1/org/hsqldb/error/ErrorCode.java
			switch(constraint.get().getErrorCode()){
			case -104:	return "Duplicado";
			case -8: return "Tiene dependencias";
			}
			return "Duplicado";
		}else{
			return "Error desconocido hable con el administrador";
		}
   }

	@ResponseStatus(HttpStatus.CONFLICT)  // 409
   @ExceptionHandler(InvalidDataAccessApiUsageException.class)
   public String handleConflictNoHayReferencias(InvalidDataAccessApiUsageException
e) {
		//Throwable cause = e.getCause();
		Optional<TransientPropertyValueException> causeTransientoProper-
tyValue = ExceptionUtils.getInstancia().buscarCause(e,TransientPropertyValueExcep-
tion.class);
		if(causeTransientoPropertyValue.isPresent()){
	return "No existe la fererencia del campo ("+causeTransientoPropertyVa-
lue.get().getPropertyName()+")";
		}else{
```

```
                        return "Error desconocido hable con el administrador";
                }
    }

        @ResponseStatus(HttpStatus.CONFLICT)  // 409
    @ExceptionHandler(PersistenceException.class)
    public String handleConflictCreado() {
                //VndErrors;
        return "El registro ya fue creado";
    }

}
```

Configuración del proyecto.

Ahora que tenemos configurado las diferentes partes de la parte servidora vamos a enlazar todas las configuraciones y por otro lado configurar la parte de spring MVC.

1º) Lo primero es enganchar los otros ficheros de configuración: (El de seguridad y la capa de persistencia.

```
@Import({SeguridadConfig.class,JpaConfig.class})
```

2º) Le vamos a indicar que active las configuraciones de spring MVC,

```
@EnableWebMvc
```

Nota: heredera ***WebMvcConfigurerAdapter***

3º) Lo siguiente que tenemos que hace indicarle las rutas donde se encuentra los beans de spring:

- Controladores: org.dbp.controller
- Dao: org.dbp.dao
- Service: org.dbp.service

```
@ComponentScan(
            basePackages={
                        "org.dbp.controller"    // Es donde se ubicaran
los controladores
                        ,"org.dbp.dao"          // Los dao.
                        ,"org.dbp.service"            // Los servicios.
                        }
            )
```

4º) Vamos a configurar el interceptor para el ententyManager por la carga perezosa.

```
@Autowired
```

```
	private LocalContainerEntityManagerFactoryBean containerEntityMan-
agerFactoryBean;

	@Override
	public void addInterceptors(final InterceptorRegistry registry) {
		super.addInterceptors(registry);
		registry.addInterceptor(new LogInterceptor());
		OpenEntityManagerInViewInterceptor interceptor = new
OpenEntityManagerInViewInterceptor();
		interceptor.setEntityManagerFactory(containerEntityManager-
FactoryBean.getObject());
		registry.addWebRequestInterceptor(interceptor); // Es para el
lazy, en los json, a la hora de parsear.
	}
```

5º) Configurar los JSP, que va a ser dentro de /WEB-INF/pages/ que es la carpeta donde pondremos los JSP.

```
    @Bean
    public InternalResourceViewResolver getInternalResourceViewResolver()
{
        final InternalResourceViewResolver resolver = new InternalRe-
sourceViewResolver();
        resolver.setPrefix("/WEB-INF/pages/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
```

6º) Configurar los recursos estáticos.

```
@Override
	public void addResourceHandlers(final ResourceHandlerRegistry reg-
istry) {
		final String entorno=env.getActiveProfiles()[0];
		registry
		.addResourceHandler("/resources/**")
		.addResourceLocations("/resources/")
		.addResourceLocations("/WEB-INF/cliente/resources/")
		.setCachePeriod(3600)
		.resourceChain(true)
		.addResolver(new PathResourceResolver());
		registry
		.addResourceHandler("/env/**")
		.addResourceLocations("/WEB-INF/cliente/resources/env/"+en-
torno+"/")
		.setCachePeriod(3600)
		.resourceChain(true)
		.addResolver(new PathResourceResolver());
		registry
		.addResourceHandler("/node_modules/**")
		.addResourceLocations("/WEB-INF/cliente/node_modules/")
		.setCachePeriod(3600)
		.resourceChain(true)
		.addResolver(new PathResourceResolver());
		registry
```

```
            .addResourceHandler("/app/**")
            .addResourceLocations("/WEB-INF/cliente/app/")
            .setCachePeriod(3600)
            .resourceChain(true)
            .addResolver(new PathResourceResolver());
          registry
            .addResourceHandler("/systemjs/**")
            .addResourceLocations("/WEB-INF/cliente/systemjs/")
            .setCachePeriod(3600)
            .resourceChain(true)
            .addResolver(new PathResourceResolver())
            ;


      }
```

7º) Configurar los parseadores de fechas, con el formato “dd/MM/yyyy” siguiendo el estándar JSR310Module

```
      @SuppressWarnings("serial")
      private  class LocalDateDeserializerEs  extends LocalDateDeserial-
izer{
            private LocalDateDeserializerEs(){
                  super(DateTimeFormatter.ofPattern("dd/MM/yyyy"));
            }
      }

      @SuppressWarnings("serial")
      private class LocalDateSerializerEs extends LocalDateSerializer{
            private  LocalDateSerializerEs(){
                  super(false,DateTimeFormatter.ofPattern("dd/MM/yyyy"));
            }
      }

      @SuppressWarnings("serial")
      private class JSR310ModuleEs extends SimpleModule{
            public JSR310ModuleEs() {
                  super(PackageVersion.VERSION);
              addDeserializer(LocalDate.class,new LocalDateDeserializ-
erEs()); //NOPMD
                addSerializer(LocalDate.class, new LocalDateSerializ-
erEs()); //NOPMD
            }
      }
```

8º) Configurar el parseador de JACKSON, con el tema de las fechas.

```
      @Override

      public void configureMessageConverters(
                  final List<HttpMessageConverter<?>> converters) {
            super.configureMessageConverters(converters);
            final Jackson2ObjectMapperBuilder builder = new Jackson2Ob-
jectMapperBuilder();
```

```
            builder
              .indentOutput(true)
              .dateFormat(new SimpleDateFormat("dd-MM-yyyy") // NOPMD (Por
que es una configuración y no procede).
              );
            converters.add(new MappingJackson2HttpMessageConverter(
                                    builder
                                          .build()
                                          .registerModule(new JSR310Modu-
leEs())));
            converters.add(new MappingJackson2XmlHttpMessageConverter(
                                    builder.createXmlMapper(true)
                                          .build()
                                          .registerModule(new JSR310Modu-
leEs())));
      }
```

9º) Registrar el controlador para la página del login.

```
      @Override
      public void addViewControllers(final ViewControllerRegistry regis-
try) {
            super.addViewControllers(registry);
            registry.addViewController("/login").setViewName("login");
            registry.setOrder(Ordered.HIGHEST_PRECEDENCE);

      }
```

10º) Activar nuestro log de aspectos

```
@EnableAspectJAutoProxy

      @Bean
      public LogAspect logAspect(){
            return new LogAspect();
      }
```

Aspectos

¿Qué es la programación orientada a aspectos?

Los aspectos nos pueden ayudar a modularizar las preocupaciones transversales. Básicamente, una preocupación transversal puede definirse como cualquier funcionalidad que afecta a varios puntos de una aplicación. Por ejemplo, la seguridad es una preocupación transversal, ya que varios métodos de una aplicación pueden contar con normas de seguridad aplicados a éstos.

Los aspectos nos ofrecen una alternativa para la herencia y la delegación que pueden ser mucho más sencillos en muchos casos. La AOP le permite definir la funcionalidad común en

una ubicación y definir, de forma declarativa, como y donde se va a aplicar, sin que tenga que modificar la clase a la que va a aplicar esta nueva característica.

Una cosa con la que tenemos que tener cuidado es que los aspectos, son para funcionalidades transversales que estén muy claras, ya que los aspectos es un código que se ejecuta de manera transversal y es muy difícil de seguir en un mantenimiento, si no sabemos que esta hay. En resumen, es una mala práctica usarlos para la lógica de la aplicación.

Ahora que sabemos que hacen magia, pero que mal usados nos puede crear un problema del copón, para que los vamos a usar:

- La transacionalidad, en este caso es Spring el que tiene configurado los aspectos.
- La seguridad global de la aplicación.
- Montar un sistema de logs, genérico.
- Y cualquier funcionalidad transversal pura que tengamos muy claro el alcance.

El logs de los aspectos

OBJETIVO

La idea es que el sistema nos saque ciertas trazas de entrada, salida y errores de un método:

Los métodos que nos interese traquear le vamos a aplicar la anotación, @DbpLog. De esta manera indicamos que métodos nos interesan que se tracen.

Por otro lado, si se produce una excepción, en un controlador, vamos a pintar una traza con los parámetros del método este nos permite no solo saber dónde se ha producido el error, sino que además nos muestra los argumentos del método en el que ha fallado.

Con este por un lado conseguimos, traquear los métodos escribiendo una anotación y por otro lado cuando se produce un error, el sistema nos informa, de los datos que había en ese momento para que lo podamos estudiar.

IMPLEMENTACIÓN

Lo hemos implementado en ***LogAspect***, que tendremos todo la anterior declarado.

```
package org.dbp.conf.aop.log;

import java.util.Arrays;
```

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
/**
 *
 * Configuración de los los aspecto para el sistema de logs.
 *
 *        - Por un lado logeamos los métodos que esten anotados por @DpbLog.
 *  - Y Siempre que tengamos una excepción, este o no anotado en los siguientes paque-
tes: org.dbp.controller
 *
 * @author david
 *
 */
@Aspect
public class LogAspect {

        private static Logger logger=LoggerFactory.getLogger(LogAspect.class); //NOPMD
        /**
         *
         * Antes de ejecutar el método.
         *
         * @param joinPoint    Es el punto de union.
         */
        @Before("execution(* *(..)) && @annotation(org.dbp.conf.aop.log.DbpLog)")
        public void antes(final JoinPoint joinPoint){
                logger.debug(" Antes: [{}] argumentos [{}]"
                                ,joinPoint.getSignature().toString()
                                ,Arrays.toString(joinPoint.getArgs()));
        }
        /**
         *
         * Despues de ejecutar un método que tiene return.
         *
         * @param joinPoint    Es el punto de union.
         * @param valdev       El valor devuelto por el método.
         */
        @AfterReturning(
                        pointcut="execution(* *(..)) && @annota-
tion(org.dbp.conf.aop.log.DbpLog)"
                        ,returning="valdev"
                        )
        public void despuesReturn(final JoinPoint joinPoint,final Object valdev){
                logger.debug(" [{}] valdev: [{}]"
                                        ,joinPoint.getSignature().toString()
                                        ,valdev);
        }
        /**
         *
```

```
	 * Despues de ejecutar un metodo que tiene void.
	 *
	 * Nota: Se ejecutara si procede, despuest de la traza de debug.
	 *
	 * @param joinPoint Es el punto de union.
	 *
	 */
	@After("execution(* *(..)) && @annotation(org.dbp.conf.aop.log.DbpLog)")
	public void despues(final JoinPoint joinPoint){
		logger.debug(" Despues: [{}] argumentos [{}]"
				,joinPoint.getSignature().toString()
				,Arrays.toString(joinPoint.getArgs()));
	}
	/**
	 *
	 * Se ejecutara siempre que tengamos una exceptión, para el paquete controller.
	 *
	 * @param joinPoint	Es el punto de union.
	 * @param exception		La excpeción que se ha ejecutado.
	 */
	@AfterThrowing(
			pointcut="execution(* org.dbp.controller.*.*(..))  "
			,throwing="exception"
			)
	public void despuesExcepcion(final JoinPoint joinPoint,final Throwable exception){
		logger.warn(" [{}] argumentos [{}] "
				,joinPoint.getSignature().toString()
				,Arrays.toString(joinPoint.getArgs()));
		logger.error(" Error [{}] "
				,joinPoint.getSignature().toString()
				,exception);
	}

}
```

Hibernate

Es una herramienta de ORM, para la plataforma Java, que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permite establecer estas relaciones.

Mapeo objeto-relacional (ORM)

Es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base datos relacional como motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional.

III Aplicación práctica

Captura de requisitos
Especificación de requisitos

Se va implementar una aplicación web de ejemplo, encargada de gestionar por un lado los asientos contables, por otro lado, un directorio de personas físicas y sus datos de contacto. La aplicación al pertenecer a una intranet de una empresa su acceso estará securizado, tendremos dos tipos de roles:

- Administrativo: Podrán realizar las siguientes tareas.
    - Registrar una persona física.
    - Consultar las personas físicas.
    - Eliminar y mantener las personas físicas.
    - Añadir, modificar o eliminar los datos de contactos vinculados a una persona física.
- Contable: Son los encargados de gestionar las cuentas contables y los asientos.
    - Crear o eliminar una cuenta contable.
    - Consultar las cuentas contables.
    - Gestionar los asientos:
        - Crear un asiento contable.
        - Modificar un asiento contable.
        - Eliminar un asiento contable.

Casos de uso

Diagrama generar de casos de uso:

Aquí tenemos que poner un diagrama general con las 3 gestiones.

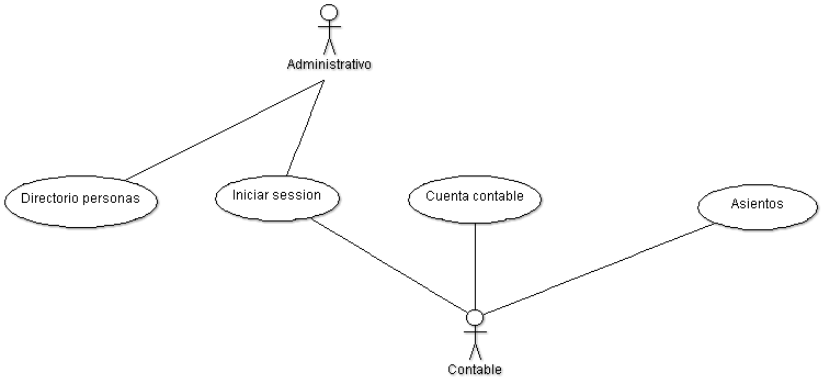


Diagrama de casos de uso para iniciar sesión

La especificación de requisitos para este grupo funcional se define a continuación.

- El sistema pedirá el usuario contraseña, en caso de ser correcto el sistema nos llevará a la página de inicio.

Nombre:	Iniciar sesión
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite validar a un usuario en el sistema	
Actores: Usuario administrativo, contable	
Precondiciones: El usuario tiene que estar dado de alta en el sistema y no esta autenticado en el sistema.	
Flujo normal: 1. El actor accede al sistema. 2. El sistema muestra una caja de texto para introducir el usuario y la contra-seña 3. El actor introduce su usuario y contraseña y pulsa sobre el botón de Enviar. 4. El sistema comprueba la validez de los datos y crea una sesión. 5. El sistema redirige al usuario a la página de inicio de la aplicación.	
Flujo alternativo: 2. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa al actor de ello permitiéndole que los corrija.	
Pos condiciones: El identificador, nombre y el role en el sistema.	

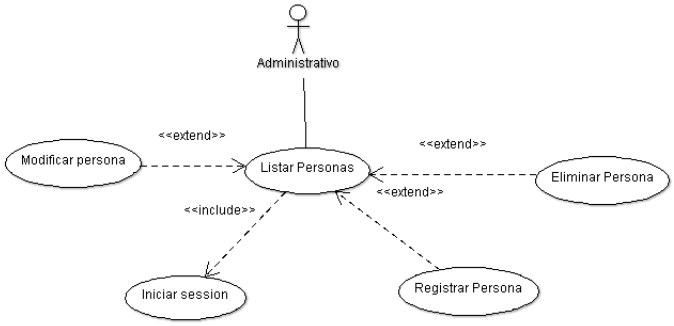
Nombre:	Inicio
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite validar a un usuario en el sistema	
Actores: Usuario administrativo, contable	
Precondiciones: El usuario tiene que estar dado de alta y autenticado.	
Flujo normal: 1. El sistema carga los datos de inicio de la aplicación.	
Flujo alternativo: No aplica.	
Pos condiciones: No aplica.	

Diagrama de casos de uso para la gestión de persona física.

La especificación de requisitos para este grupo funcional se define a continuación.

- El usuario administrativo puede consultar las personas físicas registradas en el sistema y poder filtrar los datos
- El usuario administrativo puede modificar, eliminar y registrar nuevas personas físicas y sus datos de contacto

- Para poder realizar estas operaciones es necesario que el usuario este registrado y validado por el sistema.



Nombre:	Listar personas
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite visualizar un listado de un conjunto de personas filtradas, por los siguientes criterios: • Id • Identificador Fiscal • Nombre • Apellidos	
Actores: Usuario administrativo	
Precondiciones: El usuario tiene que estar dado de alta y autenticado.	
Flujo normal: 2. El sistema muestra las diferentes cajas de texto, para introducir los filtros: (id, identificador fiscal, nombre, apellidos). 3. El actor establece los valores de filtrado en las cajas de texto, que necesiten. 4. El sistema muestra una lista con las personas que cumplen los criterios de filtro establecidos.	
Flujo alternativo: 3. Si no hay datos, mostraremos un mensaje indicado que no hay registros	
Pos condiciones: No aplica	

Nombre:	Registrar persona
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite dar de alta una persona física.	
Actores: Usuario administrativos	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar personas.	
Flujo normal: 1. El actor pulsa el botón de crear. 2. El sistema muestra un selector, para indicar el tipo de identificador fiscal (DNI,CIF), muestra las cajas de texto para introducir los datos de la persona física (Identificador fiscal, Nombre, Apellidos). 3. El sistema nos muestra una tabla con las cajas de texto, de los datos de contacto (teléfono, nombre, dirección de correo, dirección, municipio), 4. El actor introduce los datos de la persona física, los diferentes datos de contacto y pulsa el botón crear. 5. El sistema le pedirá al usuario confirmación. 6. El usuario pulsa el botón SI 7. El sistema comprueba la validez de los datos y los almacena. 8. El sistema automáticamente pasa al caso de uso modificar persona, cargando los datos de la persona registrada.	
Flujo alternativo: 4. Si el usuario pulsa Ctrl+enter en un campo de datos de contacto, el sistema crear una nueva línea de contacto. 6. El usuario pulsa No y no se crea el usuario. 7. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa el actor de ello permitiéndole que los corrija.	
Pos condiciones: Se crea un identificador para los datos de la persona física y los datos introducidos junto con este identificador quedan almacenados en una base de datos.	

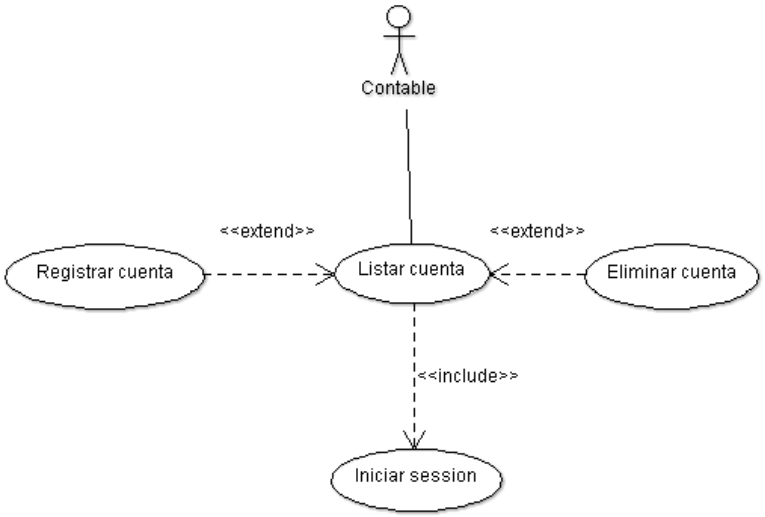
Nombre:	Eliminar persona
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite eliminar permanente, una persona física del sistema.	
Actores: El usuario administrativo	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar personas.	
Flujo normal: 1. El actor pulsa sobre el botón eliminar. 2. El sistema pregunta al actor si lo desea eliminar. 3. El actor pulsa el botón Si. 4. El sistema eliminar la persona física de forma permanente. 5. El sistema elimina el registro del listado de pantalla.	
Flujo alternativo: 1. El botón eliminar, también se puede ejecutar desde el caso de usar modificar. 3. El pulsa el botón No y no se elimina la persona física. 5. Si estamos en el caso de uso modificar, el sistema nos devuelve al caso de uso de crear.	
Pos condiciones: Quedan eliminados en la B.D todos los datos eliminados con el cliente.	

Nombre:	Modificar persona
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite modificar los datos de una persona física.	
Actores: Usuario administrativos	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar personas.	
Flujo normal: 1. El actor hace click sobre la persona física que va a modificar. 2. El sistema muestra un selector, para indicar el tipo de identificador fiscal (DNI,CIF), muestra las cajas de texto para introducir los datos de la persona física (Identificador fiscal, Nombre, Apellidos). 3. El sistema nos muestra una tabla con las cajas de texto, de los datos de contacto (teléfono, nombre, dirección de correo, dirección, municipio), 4. El sistema cargara los datos de la persona física. 5. El actor introduce los datos de la persona física, los diferentes datos de contacto y pulsa el botón crear. 6. El sistema le pedirá al usuario confirmación. 7. El usuario pulsa el botón SI 8. El sistema comprueba la validez de los datos y los almacena. 9. El sistema automáticamente pasa al caso de uso modificar persona, cargando los datos de la persona registrada.	
Flujo alternativo: 1. También puede entrar en esta operativa una vez creado correctamente una persona física. 5. Si el usuario pulsa Ctrl+enter en un campo de datos de contacto, el sistema crear una nueva línea de contacto. 7. El usuario pulsa No y no se crea el usuario. 8. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa el actor de ello permitiéndole que los corrija.	
Pos condiciones: Se guardan los datos modificados de la persona física, asociados al identificador, en la B.D.	

Diagrama de uso para las cuentas contables.

La especificación de requisitos para este grupo funcional se define a continuación.

- El usuario contable puede consultar las cuentas contables registradas en el sistema y poder filtrar los datos
- El usuario contable puede eliminar y registrar nuevas cuentas contables.
- Para poder realizar estas operaciones es necesario que el usuario este registrado y validado por el sistema.



Nombre:	Listar cuenta
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite visualizar un listado de cuenta contables filtradas, por los siguientes criterios: • Id • Descripción	
Actores: Usuario contable	
Precondiciones: El usuario tiene que estar dado de alta y autenticado.	
Flujo normal: 1. El sistema muestra las diferentes cajas de texto, para introducir los filtros: (id, descripción). 2. El actor establece los valores de filtrado en las cajas de texto, que necesiten. 3. El sistema muestra una lista con las cuentas contables que cumplen los criterios de filtro establecidos.	
Flujo alternativo: 4. Si no hay datos, mostraremos un mensaje indicado que no hay registros	
Pos condiciones: No aplica	

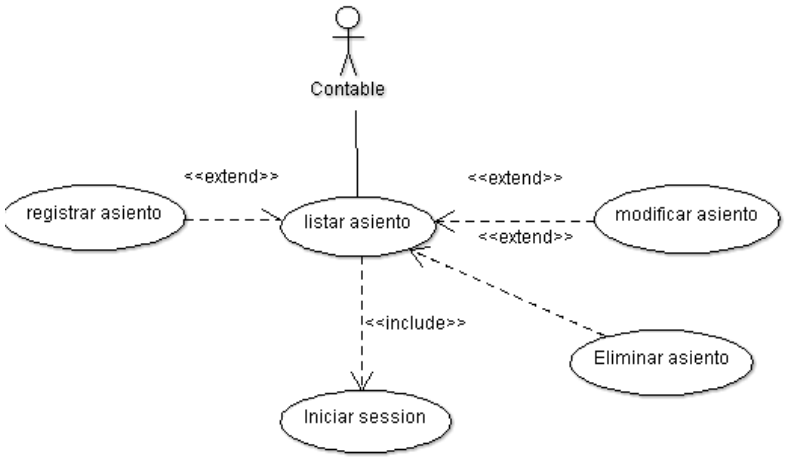
Nombre:	Registrar persona
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite dar de alta una cuenta contable	
Actores: Usuario contable	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar cuenta contable.	
Flujo normal: 1. El actor introduce los datos de la cuenta contable 2. El sistema le pedirá al usuario confirmación. 3. El usuario pulsa el botón SI 4. El sistema comprueba la validez de los datos y los almacena. 5. El sistema automáticamente crea una fila en listado con los nuevos datos.	
Flujo alternativo: 3. El usuario pulsa No y no se crea el usuario. 4. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa el actor de ello permitiéndole que los corrija.	
Pos condiciones: Se crea un identificador para los datos de la cuenta contable y los datos introducidos junto con este identificador quedan almacenados en una base de datos.	

Nombre:	Eliminar persona
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite eliminar permanente, una cuenta contable del sistema.	
Actores: El usuario contable	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar cuentas contables.	
Flujo normal: 1. El actor pulsa sobre el botón eliminar de la cuenta que desea eliminar. 2. El sistema pregunta al actor si lo desea eliminar. 3. El actor pulsa el botón Si. 4. El sistema elimina la cuenta contable de forma permanente. 5. El sistema elimina el registro del listado de pantalla.	
Flujo alternativo: 3. El pulsa el botón No y no se elimina la persona física.	
Pos condiciones: Quedan eliminados en la B.D todos los datos de la cuenta contable	

Diagramas de uso, para los asientos contables.

La especificación funcional de requisitos para este grupo funcional, de define a continuación.

- El usuario contable puede consultar los asientos contables registrados en el sistema y poder filtrar los datos.
- El usuario contable puede eliminar y registrar nuevos asientos contables.
- El usuario contable puede modificar la información de un asiento contable.
- Para poder realizar estas operaciones es necesario que el usuario este registrado y validado por el sistema.



Nombre:	Listar asientos
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite visualizar un listado de asientos contables filtradas, por los siguientes criterios: • Id • Descripción • Cuenta • Concepto	
Actores: Usuario contable	
Precondiciones: El usuario tiene que estar dado de alta y autenticado.	
Flujo normal: 1. El sistema muestra las diferentes cajas de texto, para introducir los filtros: (id, descripción, cuenta, concepto). 2. El actor establece los valores de filtrado en las cajas de texto, que necesiten y hace click en el botón de consultar. 3. El sistema muestra una lista con los asientos contables que cumplen los criterios de filtro establecidos.	
Flujo alternativo: 2. Si no hay datos, mostraremos un mensaje indicado que no hay registros	
Pos condiciones: No aplica	

Nombre:	Registrar un asiento contable
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite dar de alta un asiento contable	
Actores: Usuario contable	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar asientos.	
Flujo normal: 1. El actor pulsa el botón de crear. 2. El sistema muestra las cajas de texto para introducir los datos del asiento (descripción). 3. El sistema nos muestra una tabla con las cajas de texto, de las líneas del asiento (cuenta contable, tipo de movimiento contable, importe, concepto), 4. El actor introduce los datos del asiento contable y, las líneas del asiento y pulsa el botón crear. 5. El sistema le pedirá al usuario confirmación. 6. El usuario pulsa el botón SI 7. El sistema comprueba la validez de los datos y los almacena. 8. El sistema automáticamente pasa al caso de uso modificar persona, cargando los datos de la persona registrada.	
Flujo alternativo: 4. Si el usuario pulsa Ctrl+enter en un campo de la línea del asiento, el sistema crear una nueva línea. 6. El usuario pulsa No y no se crea el usuario. 7. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa el actor de ello permitiéndole que los corrija.	
Pos condiciones: Se crea un identificador para los datos des asiento contable y los datos introducidos junto con este identificador quedan almacenados en una base de datos.	

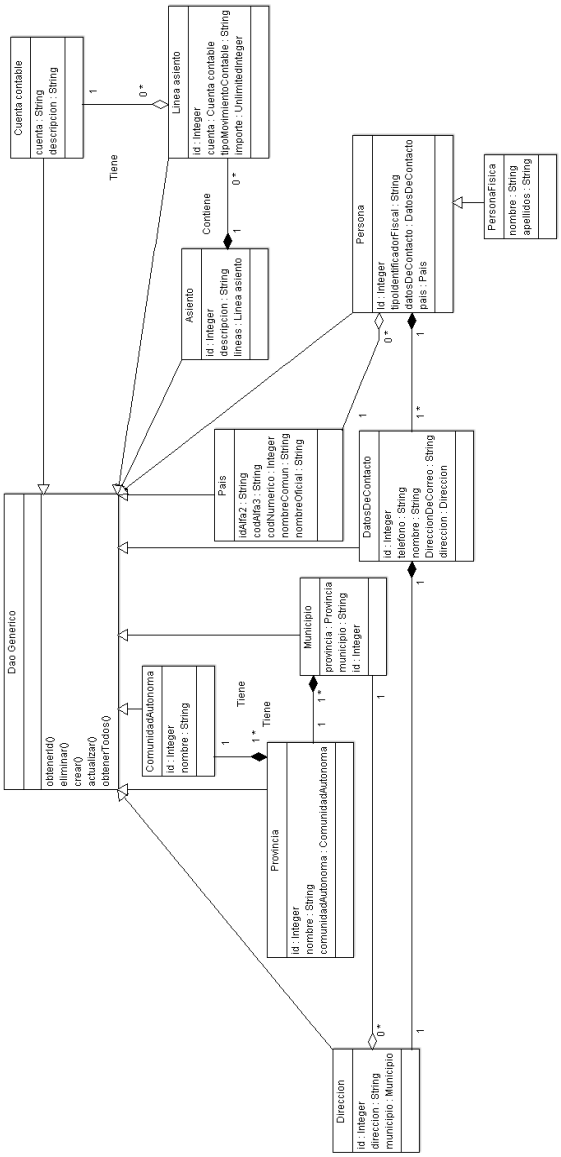
Nombre:	Eliminar un asiento contable
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite eliminar permanente, un asiento contable del sistema.	
Actores: El usuario contable	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar personas.	
Flujo normal: 1. El actor pulsa sobre el botón eliminar. 2. El sistema pregunta al actor si lo desea eliminar. 3. El actor pulsa el botón Si. 4. El sistema eliminar el asiento contable de forma permanente. 5. El sistema elimina el registro del listado de pantalla.	
Flujo alternativo: 1. El botón eliminar, también se puede ejecutar desde el caso de usar modificar. 3. El pulsa el botón No y no se elimina la persona física. 5. Si estamos en el caso de uso modificar, el sistema nos devuelve al caso de uso de crear.	
Pos condiciones: Quedan eliminados en la B.D todos los datos eliminados con el cliente.	

Nombre:	Modificar Asiento contable
Autor:	David Blanco París
Fecha:	05/08/2016
Descripción: Permite modificar los datos de un asiento contable	
Actores: Usuario contable	
Precondiciones: El usuario debe estar dado de alta en el sistema y autenticado, además se debe haber ejecutado primero el caso de uso listar asientos.	
Flujo normal: 1. El actor hace click sobre el asiento contable que va a modificar. 2. Muestra las cajas de texto para introducir los datos de la persona física (Descripción). 3. El sistema nos muestra una tabla con las cajas de texto, de la linea (cuenta contable, tipo de movimiento contable, importe, concepto), 4. El sistema cargara los datos del asiento contable. 5. El actor introduce los datos del asiento contable, los diferentes datos de contacto y pulsa el botón modificar. 6. El sistema le pedirá al usuario confirmación. 7. El usuario pulsa el botón SI 8. El sistema comprueba la validez de los datos y los almacena. 9. El sistema automáticamente pasa al caso de uso modificar asiento contable, cargando los datos de la persona registrada.	
Flujo alternativo: 1. También puede entrar en esta operativa una vez creado correctamente una persona física. 5. Si el usuario pulsa Ctrl+enter en un campo de datos de la línea, el sistema creara una nueva línea. 7. El usuario pulsa No y no se crea el usuario. 8. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa el actor de ello permitiéndole que los corrija.	
Pos condiciones: Se guardan los datos modificados del asiento contable, asociados al identificador, en la B.D.	

Análisis

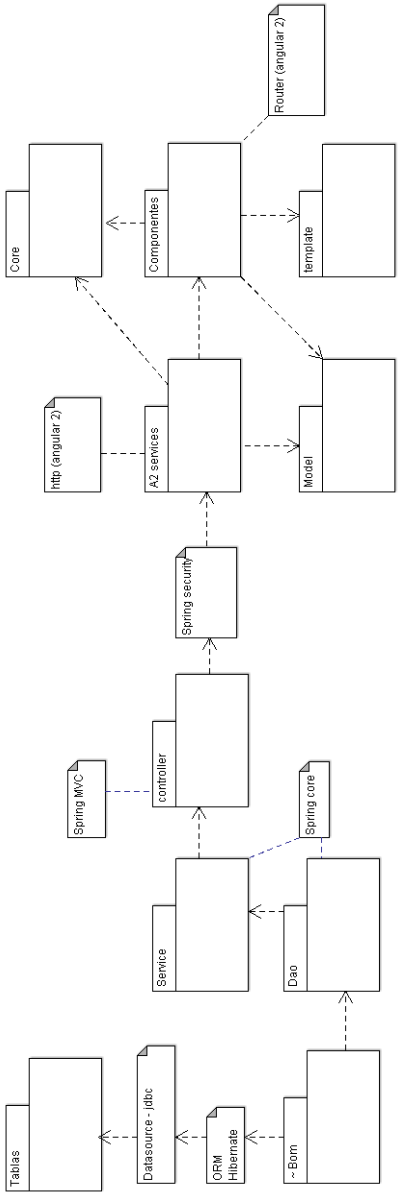
Diagrama de clases.

De la especificación de requisitos se deduce el siguiente diagrama de clases.



Diseño

Diagrama de componentes



A continuación, se describe el contenido de cada paquete:

- **Tablas**: Este paquete contiene todas las tablas incluidas en la base de datos.
- **Bom**: Este paquete, tendremos las entidades de JPA, que se vincularan directamente con las tablas de la B.D, más las vistas necesarias para las consultas.
- **Dao**: Este paquete contiene las clases que implementan el patrón dao (Objeto de acceso a datos). Un dao es un componente de software que suministra una interfaz común entra la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.
- **Service**: Este paquete contiene las clases donde tendremos la lógica de negocio de nuestra aplicación.
- **Controller**: Aquí estarán los controladores, que implementan el patrón RestFull. Aparte implementara más operaciones que el RestFull si es necesario.
- **A2 services**: Aquí estarán los servicios, que utilizara angular 2, para por un lado conectar con el servidor apoyándose en el módulo http, como servicios de ámbito general.
- **Model**: Representa los modelos, de los diferentes componentes por un lado y los objetos del servidor (Que en muchos casos van a coincidir).
- **Componentes**: Aquí están definidos los componentes, de nuestra aplicación, como las pantallas, los componentes comunes ...
- **Template**: Aquí es donde se encuentran los templates, de los componentes, como buenas practicas esos templates los tendremos en la misma ruta que el componente.
- **Core**: Aquí tendremos las clases que se encarguen de las partes genéricas de nuestra aplicación cliente, como mensajería, ventana modal.

IV Conclusiones

Conclusiones

Este proyecto me ha servido para aprender un nuevo framework como angular 2 y ver cómo integrarlo con tecnologías ya asentadas en el mercado como spring + hibernate.

Elige utilizar angular 2, en la capa cliente, para aprender las nociones en las que se basan los nuevos framework que están nacido a partir de la especificación de JavaScript 6 y JavaScript 7. Y por otro lado ver cómo integrarlo con tecnologías java 8 + spring + hibernate (En la parte servidora). También he aprendido mucho de desarrollar un proyecto sobre una beta y ver toda su evolución hasta la primera R.C. Aquí ves cómo se van tomando ciertas decisiones de diseño y como van naciendo o cambiando alguno de los módulos.

Realmente ahora tenemos que desarrollar 2 aplicaciones, por un lado, la parte servidora en una tecnología determinada y la parte cliente que ahora debido a que pretendemos realizar aplicaciones ricas en el navegador han crecido en tamaño lo cual exige nuevas tecnologías y trasladar ciertos estándares para el desarrollo de grandes de aplicaciones a la capa cliente.

También me ha servido como experiencia de ir adaptando una aplicación a los cambios de un framewors que está naciendo, en el siguiente enlace podemos ver los cambios que se han ido realizando sobre la evolución del proyecto (https://github.com/blancoparis-tfc/tfcContabilidad/issues?q=is%3Aissue+is%3Aclosed+label%3A%22COMP%3A+Migrar+angular+2%22) (Cada una de las tareas tiene un enlace a los cambios que se han realizado en el control de versiones.

Futuras mejoras

En futuro estaría bien:

- Crear un directorio de usuarios, para crear perfiles y roles en la aplicación.
- Crear un registro de facturas asociado a nuestro directorio de clientes, y una vez gestionadas las facturas se vincularán a los asientos contables.
- Seguir actualizando las versiones de angular 2 según va a evolucionando el frame-work.
- Adaptar la aplicación a angular 2 – material que está actualmente en proceso de desarrollo y no es estable, para tener una librería para trabajar con la UI.

V Herramientas de desarrollo y material de entrega

Para la implementación de esta aplicación web he utilizado las siguientes herramientas_

- ***Eclipse Mars***: Es un entorno de desarrollo libre y gratuito pensado principalmente para programar en java y nos ofrece una gran facilidad a la hora de integrar aplicaciones web, con repositorio como git y herramientas de construcción como gradle.
- ***Atom***: Es un nuevo editor, que está pensado para trabajar con aplicaciones en JavaScript y TypeScript. Se integra muy bien con estas tecnologías.
- ***Gradle***: Es una herramienta de software para la gestión y construcción de proyectos java. Lo utilizaremos para la construcción de nuestro proyecto.
- ***Node***: Es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello basado en el lenguaje de programación ECMAScript. En nuestro caso lo vamos a utilizar para la descarga de dependencias (npm) y un compilador para TypeScript (tsc).
- ***Npm***. (Node Package Manager), vamos a utilizar esta herramienta para descargar los módulos de la capa cliente, como angular 2, RxJS…
- ***Tsc***: Es la herramienta que utilizaremos para compilar los ficheros en TypeScript.
- ***TypeScript***: Es un lenguaje de programación libre y de código abierto desarrollado por Microsoft. Es un super-conjunto de JavaScript, que esencialmente añade tipado estático y objetos basados en clases.
- ***JDK 1.8***: Es un lenguaje de propósito general, concurrente y orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Es el lenguaje que usaremos en la parte servidor del proyecto.
- ***HSQLDB:*** Usaremos esta B.D embebida para la representación de los datos.
- Capa de vista:
    - Angular 2
    - RxJS
- Servidor:
    - Spring:
    - Hibernate:
- Como control de versiones y gestor de tareas hemos utilizado, el repositorio de GitHub, en la siguiente ruta: (https://github.com/blancoparis-tfc/tfcContabilidad/)

VI Bibliografía

Libros

Titulo	Autor	Isbn
Código Limpio, manual de estilo para el desarrollo ágil de software	Rober C. Martin	978-84-415-3210-6
Spring in action tercera edición	Craig Walls	978-84-415-3041-6
Patrones de diseño	Richard helm, Ralph Johnson, John Vlissides	0-201-63361-2
Real world Java EE Patterns	Adam Bien	978-1-300-14931-6

Enlaces:

Título	Enlace	Fecha
Spring framework	http://projects.spring.io/spring-framework/	01/05/2016
Angular	https://angular.io/	01/05/2016
ReactiveX	http://reactivex.io/	01/05/2016
Hibernate (ORM)	http://hibernate.org/orm/	01/05/2016
Manual del editor atom	https://atom.io/docs	01/05/2016
Blog con algunos articulos de angular 2.	http://victorsavkin.com/	01/05/2016
Spring security	http://projects.spring.io/spring-security/	01/05/2016
Teorema de cap	http://www.genbetadev.com/bases-de-datos/nosql-clasificacion-de-las-bases-de-datos-segun-el-teorema-cap	01//05/2016